phc Documentation

phc Documentation

The phc User's Manual

The phc User's Manual

Table of Contents

1. Introduction ..... **??**

..... **??**

2. Installation Instructions ..... **??**

System Requirements ..... **??**

PHP `embed SAPI` installation instructions ..... **??**

Installation Instructions ..... **??**

3. Running phc ..... **??**

Compiling executables ..... **??**

Compiling web applications ..... **??**

Alternatives ..... **??**

Writing and Reading XML ..... **??**

Internal Representations ..... **??**

Graphical Output ..... **??**

Including files ..... **??**

List of Figures

3-1. Abstract syntax tree for “Hello world”.....**??**

Chapter 1. Introduction

phc has supports limited code generation, and can be used as a front-end to parse PHP for other applications. This manual explains how to compile, install and use **phc**, how to compile command-line and web applications, and how to convert PHP to an XML reprentation, and back.

> **Note:** Documentation of the **phc** API, including how to write plugins, can be found in *The **phc** Developer's Manual*.

The documentation for this and for older versions of **phc** can be found in the archive (http://www.phpcompiler.org/src/archive/). Moreover, although we have tried to document **phc** as well as we can, if anything is still unclear, please let us know by sending an email to the mailing list (http://www.phpcompiler.org/mailinglist.html).

Chapter 2. Installation Instructions

System Requirements

> **Warning**
>
> These instructions only apply if you *don't* intend to modify phc, and you are using a downloaded **phc** release. If you intend to modify it, or you are using the **phc** SVN repository (http://code.google.com/p/phc), please refer to the instructions for developers.

phc needs a Unix-like environment to run (it has been tested on Linux, Solaris, FreeBSD, Cygwin and Mac OS X). To compile **phc**, you will need:

- `g++` *version 3.4.0* or higher
- `make`
- `Boost` *version 1.34* or higher
- `PHP5 embed SAPI` (version 5.2.x recommended; refer to *PHP embed SAPI installation instructions* for more details). This is required to compile PHP code with **phc**.
- Xerces-C++ (http://xml.apache.org/xerces-c/) if you want support for XML parsing (you don't need Xerces for XML unparsing).
- `Boehm` garbage collector is used in **phc**, but not in code compiled by **phc**. If unavailable, it can be disabled with `--disable-gc`, but **phc** will leak all memory it uses.

The following dependencies are optional:

- a DOT viewer such as graphviz (http://www.graphviz.org) if you want to be able to view the graphical output generated by **phc** (for example, syntax trees)

Under Debian/Ubuntu, the following command will install nearly all dependencies: `apt-get install build-essential libboost-dev libxerces27-dev graphviz libgc-dev`. You will still need to install the PHP embed SAPI manually.

PHP `embed SAPI` installation instructions

If you do not intend to compile PHP code using **phc**, you may skip this section.

In order to compile code, **phc** must have the PHP embed SAPI available, which is typically not available via standard package managers. The embed SAPI is also required for compiling stand-alone executables.

Download the PHP source (.tar.gz) package from php.net (http://www.php.net). We will assume you downloaded PHP version 5.2.6, the latest version available at time of writing. To extract PHP,

```
tar zxvf php-5.2.6.tar.gz
```

This will create a new directory `php-5.2.6`. In order to configure and compile PHP, you must know what configuration options you require. These are likely to be the same as the version of PHP you are currently using, which can be examined with the command

```
php -i | grep Configure
```

We will assume these options are `--enable-bcmath --with-gmp --with-mysql`, a configuration which we occasionally use for benchmarking. You are ready to build PHP. When configuring, you must add the `--enable-embed` option.

```
CFLAGS="-O3" ./configure --enable-bcmath --with-gmp --with-mysql --enable-embed
make
```

If this command does not succeed, there is a problem with PHP on your system, and you should file a bug report with the PHP group (http://bugs.php.net). (There is a known bug, and long work-around for OSX already filed in the PHP bug system).

The most important part of the command is `--enable-embed`. While the `CFLAGS="-O3"` environmental variable is optional, we find it speeds up the executable by about four times. If PHP is already installed on your system, you may want to install this version separately, using the `--prefix` option. Other configuration options are discussed in the developer manual. Finally, install the embed SAPI:

```
make install
```

Installation Instructions

First of all, you must download (http://www.phpcompiler.org/downloads.html) the latest release of **phc**. To extract **phc**,

```
tar zxvf phc-0.2.0.tar.gz
```

This will create a new directory `phc-0.2.0` that contains the **phc** source tree. Finally, you must compile **phc**. If the dependencies are in their standard locations, you should be able to simply type

```
cd phc-0.2.0
./configure
make
```

(Consult `./configure --help` for configuration options, if your dependencies are not in standard locations). This should compile without any warnings or errors. If this step fails, please send a bug report to the mailing list (http://www.phpcompiler.org/mailinglist.html) with as much information about your system as you can give, and we will try to resolve it. Finally, install **phc** using

```
make install
```

For information on running **phc**, see *Running phc*. If you can follow those instructions and you get the output you should get, congratulations! You have successfully installed **phc**.

Chapter 3. Running phc

Once you have installed **phc** (see *Installation Instructions*), run it by typing

```
phc --help
```

You should see

```
phc 0.2.0

Usage: phc [OPTIONS]... [FILES]...

  -h, --help               Print help and exit
      --full-help          Print help, including hidden options, and exit
  -V, --version            Print version and exit

GENERAL OPTIONS:
  -v, --verbose            Verbose output  (default=off)
  -c, --compile            Compile  (default=off)
      --pretty-print       Pretty print input according to the Zend style
                             guidelines  (default=off)
      --obfuscate          Obfuscate input  (default=off)
      --run=STRING         Run the specified plugin (may be specified multiple
                             times)
      --r-option=STRING    Pass option to a plugin (specify multiple flags in
                             the same order as multiple plugins - 1 option only
                             per plugin)
  -d, --define=STRING      Define ini entry (only affects -c and --include)

INPUT OPTIONS:
      --read-xml=passname  Assume the input is in XML format. Start processing
                             after the named pass
      --include            Parse included or required files at compile-time
                             (default=off)

COMPILATION OPTIONS:
  -C, --c-option=STRING    Pass option to the C compile (e.g., -C-g; can be
                             specified multiple times)
      --extension=NAME     Generate a PHP extension called NAME instead of a
                             standalone application
  -O, --optimize=STRING    Optimize  (default='0')
  -o, --output=FILE        Place executable into file FILE
  -e, --execute            Run executable after compiling (implies -c)
                             (default=off)

PRETTY PRINTING OPTIONS:
      --next-line-curlies  Output the opening curly on the next line instead of
                             on the same line  (default=off)
      --no-leading-tab     Don't start every line in between  with a
                             tab  (default=off)
      --tab=STRING         String to use for tabs while unparsing
                             (default=' ')
```

```
      --no-hash-bang       Do not output any #! lines  (default=off)
```

Now write a very small PHP script, for example

```
<? echo "Hello world!"; ?>
```

and save it to `helloworld.php`. Then run **phc**:

```
phc --pretty-print helloworld.php
```

This should output a pretty-printed version of your PHP script back to standard output:

```
<?php
   echo "Hello world!";
?>
```

You can see a list of options controlling the style of pretty printing, using the `--full-help` option.

Compiling executables

phc can compile either executables or extensions. To compile an executable, **phc** creates C code, which it compiles and links to the PHP embed SAPI. Since it links to PHP, you have access to all of PHP's large built-in standard library. In order to compile the "hello world" executable from before, run

```
phc -c helloworld.php -o helloworld
```

This creates an executable `helloworld`, which can then be run

```
./helloworld
```

If you prefer to run your executable immediately after it compiles, use the `-e`. **phc** will compile your program, then immediately execute it. You can also view the C code generated by **phc**:

```
phc --generate-c helloworld.php > helloworld.c
```

One of the advantages of **phc** is that it can optimize your program. Using the `-O` flag, you can instruct **phc** to analyse your source code, and perform simple optimizations. On simple benchmarks, this can increase the speed of your application by 50%. To optimize:

```
phc -O2 -c helloworld.php -o helloworld
```

phc generates C code, which is then compiled by `gcc`. To see the command passed to `gcc` by **phc**, use the `-v` flag.

If you specify the `-O` flag, **phc** will also pass the `-O` flag to `gcc`, which will optimize your code further. The argument to the `-O` flag must therefore be usable by `gcc`, so it must be any of `-O0` (default), `-O1`, `-O2`, `-O3` or `-Os`. Consult the gcc manual (http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html#Optimize-Options) for more details.

It is also possible to pass command-line arguments to `gcc` through **phc**, using the `-C` flag. For example, to disable inlining of the generated code by `gcc`, using `-fno-inline`:

```
phc -c -O2 helloworld.php -o helloworld -C-fno-inline
```

Compiling web applications

> **Warning**
>
> In order to compile web applications, it is currently necessary to alter your `php.ini` file, or have access to the root account. We welcome suggetions of a different method which avoids these requirements, especially if they would work in a shared hosting environment.

> **Warning**
>
> This section is experimental. Please report any problems (http://www.phpcompiler.org/mailinglist.html).

We have created the command-line option `--web-app`, which will in the future automate the process of compiling a web application. Unfortunately, for now, please follow these steps.

We describe how to create and install an extension using the C code generated by **phc**. While we give an overview of creating extensions, significantly more detail can be found in the Zend Extension Writing Tutorial (http://devzone.zend.com/node/view/id/1021) and in Extending and Embedding PHP (http://www.amazon.com/dp/067232704X).

To begin, create a new directory for the extension. We'll use `ext/` in our example. Generate C code from `helloworld.php` using **phc**.

```
phc --generate-c helloworld.php > ext/helloworld.c
```

Create a new file, `ext/config.m4`, by copying the following, and changing instances of "helloworld" appropriately:

```
PHP_ARG_ENABLE(helloworld, whether to enable Hello World support,
[ --enable-helloworld   Enable Hello World support])

if test "$PHP_HELLOWORLD" = "yes"; then
  AC_DEFINE(HAVE_HELLOWORLD, 1, [Whether you have Hello World])
  PHP_NEW_EXTENSION(helloworld, helloworld.c, $ext_shared)
fi
```

In the previous section, we described using the PHP embed SAPI. If you installed a copy of PHP with `--enable-embed` enabled, it is important *NOT* to use that version for the following commands. Instead, you should the same version as your webserver uses. From the ext/ directory, run:

```
phpize --with-php-config=/usr/bin/php-config
./configure --enable-helloworld
```

Build and install the extension (if you dont have root, refer instead to *Alternatives*):

```
make
sudo make install
```

In your web folder, replace the existing `helloworld.php` file contents with the following:

```
<?php
dl ("helloworld.so");
__MAIN__ ();
?>
```

If the `dl()` function is not enabled in your `php.ini` file, enable it:

```
enable_dl = On;
```

Accessing `helloworld.php` should now work.

Alternatives

Instead of setting `enable_dl`, you can instead load the extension manually in your `php.ini` file:

```
extension=helloworld
```

You can also avoid installing the extension using `sudo make install` by adding an alternate extension directory:

```
extensions_dir="/full/path/to/ext"
```

Writing and Reading XML

phc can output an XML representation of the PHP script. You can use this representation if you want to process PHP scripts using tools in your desired framework, instead of using **phc** plugins. After processing the XML representation, **phc** can convert it back into PHP. To generate an XML version of a PHP script, run

```
./phc --dump-xml=ast helloworld.php > helloworld.xml
```

When reading the XML back in, all the usual features of **phc** are again available; in particular, it is possible to read an XML file, and write PHP syntax. To convert the XML file we just generated back to PHP syntax, run

```
./phc --read-xml=ast --pretty-print helloworld.xml
```

The generated XML should use the schema http://www.phpcompiler.org/phc-1.0. However, our XML schema is currently broken.

Internal Representations

After parsing, **phc** converts a PHP script into an Abstract Syntax Tree (AST) (this is further explained in Chapter 3 in *The **phc** Developer's Manual*). This is very useful for processing PHP scripts which you wish to convert back into PHP. However, for some tasks, especially program analysis, a simpler form of the PHP script is more suitable. **phc** offers two other Internal Representations (IRs). The High-level Internal Representation (HIR) simplifies most expressions by assigning them to temporary variables. However, code represented in the HIR is still valid PHP. The Medium-level Internal Representation (MIR) converts HIR statements to simpler components, for example converting control-flow statements like the `for`-loop, into `goto`s. To view PHP in any of these forms, use the `--dump` option:

```
phc --dump=ast helloworld.php
phc --dump=hir helloworld.php
phc --dump=mir helloworld.php
```

Nearly all **phc** options work as well on the HIR and MIR as on the AST. For example, XML can be read and written:

```
phc --dump-xml=hir | ./myprog | phc --read-xml=hir
```

Graphical Output

If you have a DOT viewer installed on your system (for example, graphviz (http://www.graphviz.org)), you can view the AST graphically. First, ask **phc** to output the AST in DOT format:

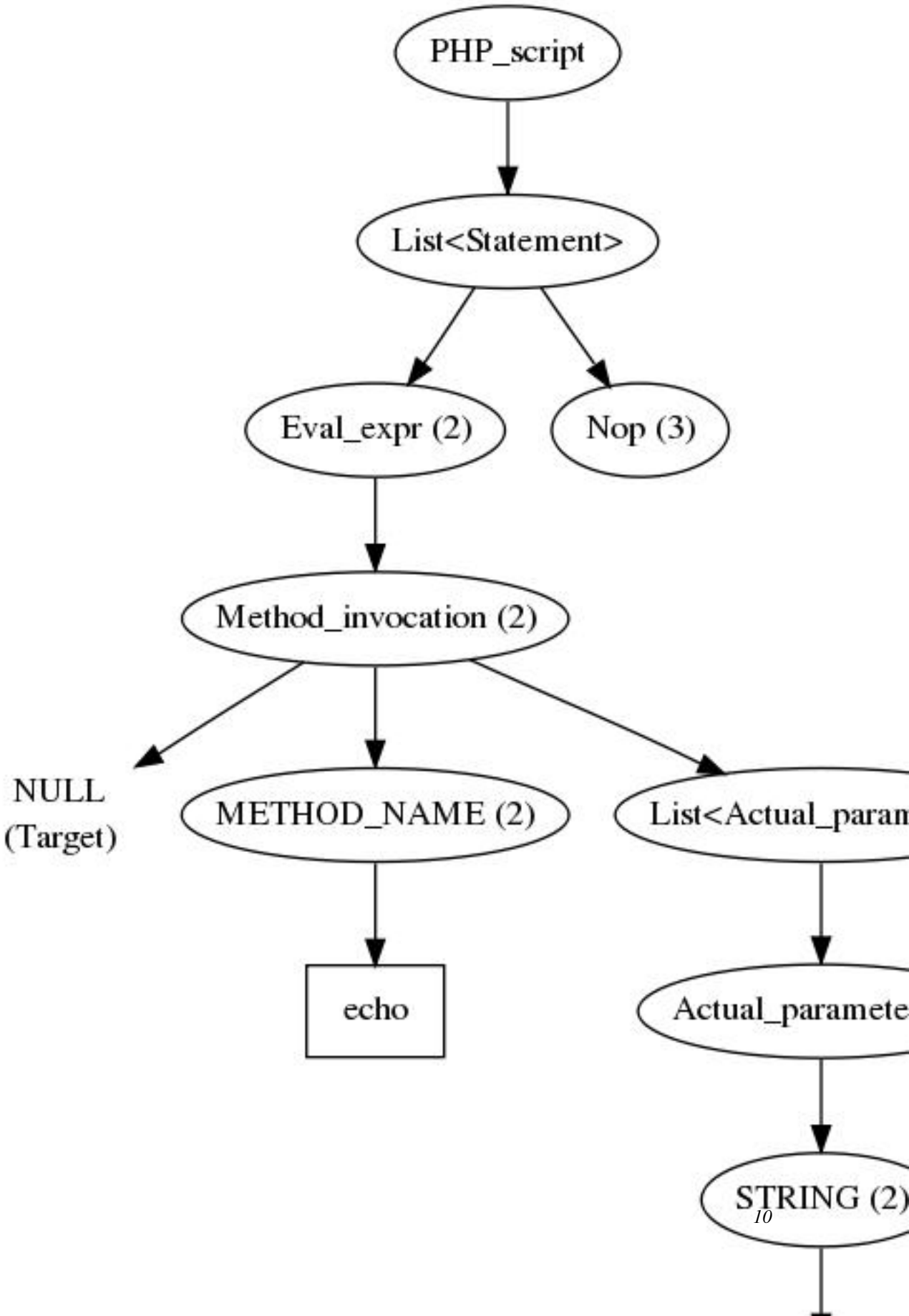
```
./phc --dump-dot=ast helloworld.php > helloworld.dot
```

You can then view the tree (`helloworld.dot`) using Graphviz. In most Unix/Linux systems, you should be able to do

```
dotty helloworld.dot
```

And you should see the tree; it should look similar to the tree shown in figure Figure 3-1.

Figure 3-1. Abstract syntax tree for “Hello world”



Including files

phc has initial support for compile-time processing of PHP's `include` built-in. Enabling this feature inserts the included statements in the AST in the place of the `include` statement. Included functions, classes and interfaces become part of the file's top-level scope. In the event that **phc** is not able to process the `include` statement (for example, if the file cannot be found), a warning is issued, and the `include` statement is left in place. To enable this support, run

```
./phc --include script_with_includes.php
```

The include support is intended to mimic PHP's include built-in (http://php.net/manual/en/function.include.php), as far as can be achieved at compile time. **phc** supports:

- Moving included statements to the point at which `include` was called. Naturally, these statement's use the variable scope at the point at which they are included,
- Preserving `__FILE__` and `__LINE__` statements,
- Moving included functions to the `%MAIN%` class, and importing the included classes,
- `include`, and `require`. If the specified file cannot be found, parsed, or if the argument to `include` is not a string literal, the include statement is left in place.

phc does not support:

- Return values in included scripts. We intend to support these in the future. They will likely be supported in a later stage of the compilation process, instead of in the AST,
- Calling `include` on anything other than a literal string containing the filename of a local file. This excludes variables and remote files. These may be supported when more static analyses are available,
- `include_once` and `require_once`, as we cannot guarantee that the file to be included is not included elsewhere. These statements will not be processed, and combinations of `include` or `require` and `include_once` or `require_once` may cause incorrect behaviour with this option set,
- Updating `get_included_files()` to reflect the included files.

The phc Developer's Manual

The phc Developer's Manual

Table of Contents

1. Introduction ..... **??**
..... **??**
System Requirements ..... **??**
Building PHP for **phc** development ..... **??**
Demonstration ..... **??**
The Source Program ..... **??**
The Transform ..... **??**
The Result ..... **??**
Writing Plugins ..... **??**
Reference ..... **??**
I. Tree Traversal API Tutorials ..... **??**
2. Getting Started ..... **??**
..... **??**
Compiling a Plugin ..... **??**
About `extern "C"` ..... **??**
Abstract Syntax ..... **??**
The Abstract Syntax Tree ..... **??**
Working with the AST ..... **??**
Actually ..... **??**
Writing Stand Alone Applications ..... **??**
What's Next? ..... **??**
3. Traversing the Tree ..... **??**
..... **??**
The Grammar (Revisited) ..... **??**
Statements and Expressions ..... **??**
The Difficult Solution ..... **??**
The Easy Solution ..... **??**
Pre and Post Methods ..... **??**
4. Modifying Tree Nodes ..... **??**
..... **??**
First Attempt ..... **??**
Modifying the Parameters ..... **??**
Refactoring ..... **??**
What's Next? ..... **??**
5. Restructuring the Tree ..... **??**
..... **??**
Introducing the `Tree_transform` API ..... **??**
The Implementation ..... **??**
Running Transformations ..... **??**
A Subtlety ..... **??**
What's Next? ..... **??**
6. Using State ..... **??**
..... **??**
The Implementation ..... **??**
What's Next? ..... **??**

7. Modifying the Traversal Order ..... ??
..... ??
The Solution ..... ??
What's Next? ..... ??
8. Returning Lists ..... ??
..... ??
Deleting Nodes ..... ??
Using the XML unparser ..... ??
The Full Transform ..... ??
What's Next? ..... ??
II. Reference ..... ??
9. The Abstract Grammar ..... ??
..... ??
Overall Structure ..... ??
Statements ..... ??
Expressions ..... ??
Additional Structure ..... ??
Mix-in Code ..... ??
10. Representing PHP ..... ??
..... ??
Variables ..... ??
Comments ..... ??
String parsing ..... ??
`elseif` ..... ??
Miscellaneous Other Changes ..... ??
Comparison to the PHP grammar ..... ??
11. Limitations ..... ??
..... ??
Comments ..... ??
Numbers ..... ??
Scopes ..... ??
Other issues ..... ??
12. Overview of the AST classes and transformation API ..... ??
..... ??
The AST classes ..... ??
Deep Equality ..... ??
Cloning ..... ??
Pattern Matching ..... ??
The Visitor API ..... ??
The Transform API ..... ??
13. `Maketea` Theory ..... ??
Introduction ..... ??
The Grammar Formalism ..... ??
Context Resolution ..... ??
Contexts ..... ??
Reducing Contexts ..... ??
Resolution for Disjunctions ..... ??

III. Development guide ..... **??**
14. Porting and Packaging ..... **??**
We need porters, packagers and maintainers ..... **??**
Packaging hints ..... **??**
Test suite ..... **??**
phc packages ..... **??**

List of Figures

1-1. Abstract syntax tree for the demo example ..... **??**
3-1. Abstract syntax tree for the running example ..... **??**
10-1. Function call in the AST ..... **??**
10-2. Function call as represented by PHP ..... **??**
12-1. Sequence Diagram for the Visitor API ..... **??**
12-2. Sequence Diagram for the Transform API ..... **??**

Chapter 1. Introduction

From the start, one of the design goals of **phc** has been to provide a useful framework for writing applications that process PHP scripts. **phc** parses PHP code into an internal representation known as an *abstract syntax tree* or *AST*. Applications can process PHP code by analysing and modifying this abstract representation in one of two ways:

- **phc** supports *plugins*. Plugins are modules that can be loaded into **phc**, which get access to the AST. **phc** provides sophisticated support for writing operations over the AST through the *Tree Transformation API*.
- Alternatively, you can export the AST to *XML*. You can then process the XML in any way you like, and then use **phc** to convert the XML back to PHP.

The *Tree Traversal API Tutorials* explain how to write plugins for **phc**, and provide numerous examples. You will find *Reference* very useful when writing serious applications using **phc**.

Although we have tried to document **phc** as well as we can, if anything is still unclear, please let us know by sending an email to the mailing list (http://www.phpcompiler.org/mailinglist.html).

System Requirements

If you want to modify the internals of **phc** in other ways than through the explicit API we provide for doing so, you will following tools listed below, in addition to those detailed in the user manual (Chapter 2 in *The **phc** User's Manual*). However, most people should not need these tools (even if you are implementing tools based on **phc**).

- `flex` if you need to modify the lexical analyser
- `bison` if you need to modify the parser
- maketea (http://www.maketea.org) if you want to modify the **phc** grammars (or the AST/HIR/MIR classes)
- gengetopt (http://www.gnu.org/software/gengetopt/gengetopt.html) if you need to add additional command line arguments; you will need version 0.20 or higher
- gperf (http://www.gnu.org/software/gperf/gperf.html) if you need to modify the list of keywords recognized by the lexical analyser

Building PHP for phc development

When compiling PHP for use with **phc**, there are a few options:

- Development: For developing **phc** or debugging **phc** problems, it is worthwhile to have debugging symbols and leak checkers enabled.

```
CFLAGS="-O0 -ggdb3" ./configure --enable-debug
--enable-maintainer-zts --enable-embed
```

- Deployment: For performance, optimization should be used:

  ```
  CFLAGS="-O3 -g" ./configure --enable-embed
  ```

- Benchmarking: In order to be fair, both **phc**-generated-code and PHP should be compiled with -O3. The are also some options required to run some benchmarks. The prefix is supplied to correspond to benchmarking scripts we provide:

  ```
  CFLAGS="-O3 -DNDEBUG" ./configure --enable-embed --enable-bcmath
  --with-gmp --prefix=/usr/local/php-opt
  ```

Demonstration

This section is intended as a quick introduction outlining what the current release of **phc** can do for you. It does not explain everything in detail.

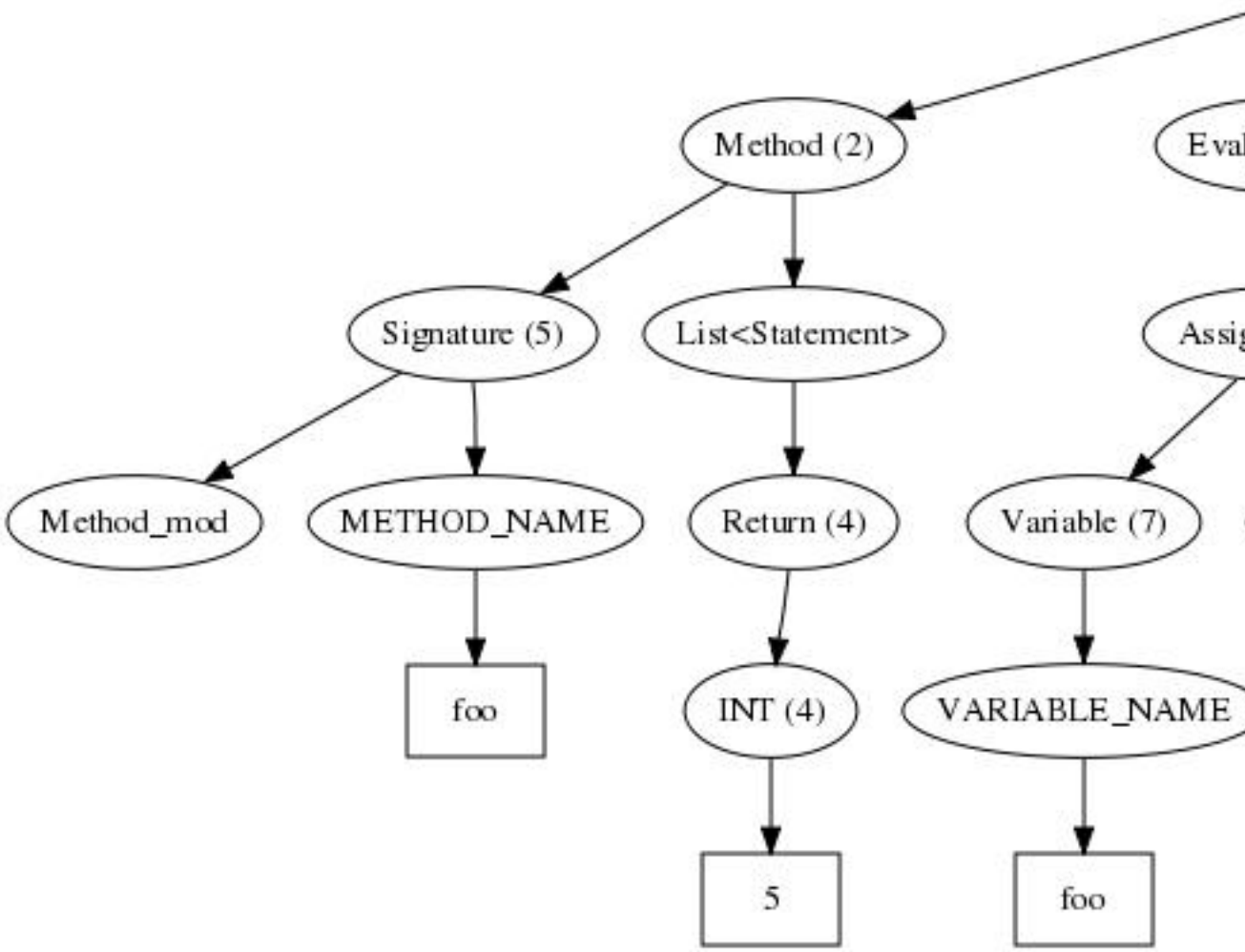
The Source Program

Consider the following simple PHP script.

```
<?php
   function foo()
   {
      return 5;
   }

   $foo = foo();
   echo "foo is $foo<br>";
?>
```

Internally this program gets represented as an abstract syntax tree, as shown in figure Figure 1-1.

Figure 1-1. Abstract syntax tree for the demo example



The Transform

Suppose we want to rename function `foo` to `bar`. This is done by the following plugin:

```
#include "AST_visitor.h"
#include "pass_manager/Plugin_pass.h"

class Rename_foo_to_bar : public Visitor
{
   void pre_method_name(METHOD_NAME* in)
   {
      if(*in->value == "foo")
         in->value = new String("bar");
   }
};

extern "C" void run_ast (AST::PHP_script* in, Pass_manager* pm, String* option)
{
    Rename_foo_to_bar f2b;
    php_script->visit(&f2b);
}

extern "C" void load (Pass_manager* pm, Plugin_pass* pass)
{
pm->add_after_named_pass (pass, new String("ast"));
}
```

The Result

Running **phc** gives

```
<?php
   function bar()
   {
      return 5;
   }

   $foo = bar();
   echo "foo is " . $foo . "<br>";
?>
```

where the name of the function has been changed, while the name of the variable remained unaltered, as has the text `"foo"` inside the string. It's that simple! Of course, in this example, it would have been quicker to do it by hand, but that's not the point; the example shows how easy it is to operate on PHP scripts within the **phc** framework.

Writing Plugins

Getting Started introduces writing plugins for **phc**. It then explains how **phc** represents PHP scripts internally, and shows how to write a first (but ultimately wrong) attempt at simple plugin that counts the number of statements in a PHP script.

Traversing the Tree introduces the support that **phc** offers for traversing (and transforming) scripts. It shows how to write a plugin that correctly counts the number of statements in a script.

Modifying Tree Nodes shows how you can modify nodes in the tree (without modifying the structure of the tree). It shows how to replace calls to `mysql_connect` by calls to `dbx_connect`.

Restructuring the Tree shows how you can modify the structure of the tree. It works through an example that removes unnecessary string concatenations (for example, `$a . ""` is replaced by just `$a`).

Using State explains an advanced features of pattern matching, and shows an important technique: the use of state in transformations (where one transformation depends on a previous transformation). It shows how to write a program that renames all functions `foo` in a script to `db_foo`, if there are calls to a database engine within `foo`.

Modifying the Traversal Order explains how to change the order in which the children of a node are visited, avoid visiting some children, or how to execute a piece of code in between visiting two children.

Returning Lists shows how to define transformations that replace nodes in the tree by multiple other nodes, and how to delete nodes from the tree. It also shows to call the **phc** parser and unparsers from plugins.

Reference

phc represents PHP scripts internally as an abstract syntax tree. The structure of this tree is dictated by the *The Abstract Grammar*. The grammar definition is a very important part of **phc**.

phc's view on the world (as dictated by the grammar) does not completely agree with the PHP standard view. *Representing PHP* describes how the various PHP constructs get translated into the abstract syntax.

Overview of the AST classes and transformation API gives an overview of the AST classes, the tree visitor API and the tree transformation API from a programmer's perspective.

`Maketea` is a tool bundled with **phc** which, based on a grammar definition of a language, generates a C++ hierarchy for the corresponding abstract syntax tree, a tree transformation and visitor API, and deep cloning, deep equality and pattern matching on the AST. *Maketea Theory* explains some of the theory behind `maketea`; in particular, the grammar formalism, the mapping from the grammar to the AST classes, and the derivation of the tree transformation API.

I. Tree Traversal API Tutorials

Chapter 2. Getting Started

For this introductory tutorial, we assume that you have successfully downloaded and installed **phc**, and that you know how to run it (Chapter 2 in *The **phc** User's Manual* and Chapter 3 in *The **phc** User's Manual*). This tutorial gets you started with using **phc** to develop your own tools for PHP by writing plugins.

Compiling a Plugin

To get up and running, we'll first write a "hello world" plugin that does nothing except print a string. Create a new directory, say `~/myplugins` and create a new file `helloworld.cpp`:

```
#include <AST.h>
#include <pass_manager/Plugin_pass.h>

extern "C" void load (Pass_manager* pm, Plugin_pass* pass)
{
   pm->add_after_named_pass (pass, new String ("ast"));
}

extern "C" void run_ast (AST::PHP_script* in, Pass_manager* pm, String* option)
{
   cout << "Hello world (I'm a phc plugin!)" << endl;
}
```

This is an example of an (almost) minimal plugin. Every plugin you write must contain these functions, with these exact signatures. `load` is run when **phc** starts, giving your plugin the opportunity to add itself to the list of passes **phc** runs. In this example, it is added after the "ast" pass. When **phc** processes a PHP script, it runs all of the passes on it in turn. When it's your plugin's turn, it calls your version of `run_ast`.

To compile the plugin, run

```
~/myplugins$ phc_compile_plugin helloworld.cpp
```

(`phc_compile_plugin` is a small shellscript that makes the task of compiling plugins easier; it calls `g++` in a platform independent way; if you're curious, you can open it in any text editor.) Finally, run the plugin using

```
~/myplugins$ phc --run helloworld.la sometest.php
```

(You need to pass in an input script to **phc** even though our plugin does not use it.) If that worked as expected, congratulations: you've just written your first **phc** plugin! :-)

About `extern "C"`

You may have been wondering what the `extern "C"` in the definition of `load` and `run_ast` is for; the reason is that **phc** uses the `libtool`'s `libltdl` interface to load your plugin; if the functions are not declared as `extern "C"`, **phc** will not be able to find them in your plugin because the name of that function will have been mangled by the C++ compiler. It does not mean that you cannot write C++ code inside these functions.

If you don't understand any of that, don't worry about it: just remember that you need to declare `load`, `run_ast`, and a small number of other functions which we'll name later, as `extern "C"` and everything will be fine. (You don't need `extern "C"` for any functions you might define).

Abstract Syntax

To be able to do anything useful in your plugins, you need to know how **phc** represents PHP code internally. **phc**'s view of PHP scripts is described by an *abstract grammar*. An abstract grammar describes how the contents of a PHP script are structured. A grammar consists of a number of rules. For example, there is a rule in the grammar that describes how `if` statements work:

```
If ::= Expr iftrue:Statement* iffalse:Statement* ;
```

This rules reads: "*An `if` statement consists of an expression* (the condition of the if-statement), *a list of statements called 'iftrue'* (the instructions that get executed when the condition holds), *and another list of statements called 'iffalse'* (the instructions that get executed when the condition does not hold)". The asterisk (`*`) in the rule means "list of".

As a second example, consider the rule that describes arrays in PHP. This rule should cover things such as `array()`, `array("a", "b")` and `array(1 => "a", 2 => "g")`. Arrays are described by the following two rules.

```
Array ::= Array_elem* ;
Array_elem ::= key:Expr? val:Expr ;
```

(Actually, this is a simplification, but it will do for the moment.) These two rules say that "*an array consists of a list of array elements*", and an "*array element has an optional expression called 'key', and a second expression called 'val'*". The question mark (`?`) means "optional". Note that the grammar does not record the need for the keyword `array`, or for the parentheses and commas. We do not need to record these, because we already *know* that we are talking about an array; all we need to know is what the array elements are.

The Abstract Syntax Tree

When **phc** reads a PHP script, it builds up an internal representation of the script. This representation is known as an *abstract syntax tree* (or AST for short). The structure of the AST follows directly from the abstract grammar. For people familiar with XML, this tree can be compared to the DOM representation of an XML script (and in fact, **phc** can output the AST as an XML document, see Chapter 3 in *The **phc** User's Manual*).

For example, consider `if`-statements again. An `if`-statement is represented by an instance of the `If` class, which is (approximately) defined as follows.

```
class If
{
public:
   Expr* expr;
   Statement_list* iftrue;
   Statement_list* iffalse;
};
```

Thus, the name of the rule (`if ::= ...`) translates into a class `If`, and the elements on the right hand side of the rule (`Expr iftrue:Statement* iffalse:Statement*`) correspond directly to the class members. The class `Statement_list` inherits from the STL `list` class, and can thus be treated as such.

Similarly, the class definitions for arrays and array elements look like

```
class Array
{
public:
   Array_elem_list* array_elems;
};

class Array_elem
{
public:
   Expr* key;
   Expr* val;
};
```

When you start developing applications with **phc** you will find it useful to consult the full description of the grammar, which can be found in Chapter 9. A detailed explanation of the structure of this grammar, and how it converts to the C++ class structure, can be found in Chapter 13. Some notes on how **phc** converts normal PHP code into abstract syntax can be found in Chapter 10.

Working with the AST

When you want to build tools based on **phc**, you do not have to understand how the abstract syntax tree is built, because this is done for you. Once the tree has been built, you can examine or modify the tree in any way you want. When you are finished, you can ask **phc** to output the tree to normal PHP code again.

Let's write a very simple plugin that counts the number of statements in a script. Create a new file `~/myplugins/count_statements.cpp`. Recall the skeleton plugin:

```
#include <AST.h>
#include <pass_manager/Plugin_pass.h>

extern "C" void load (Pass_manager* pm, Plugin_pass* pass)
{
   pm->add_after_named_pass (pass, new String ("ast"));
```

```
}

extern "C" void run_ast (AST::PHP_script* in, Pass_manager* pm, String* option)
{
}
```

You will notice that `run_ast` gets passed an object of type `PHP_script`. This is the top-level node of the generated AST. If you look at the grammar (Chapter 9), you will find that `PHP_script` corresponds to the following rule:

```
PHP_script ::= Statement* ;
```

Thus, as far as **phc** is concerned, a PHP script consists of a number of statements. The class `PHP_script` will have therefore have one member, called `statements`, the list of statements. So, to count the number of classes, all we have to do is query the number of elements in the `statements` list:

```
#include <AST.h>
#include <pass_manager/Plugin_pass.h>

extern "C" void load (Pass_manager* pm, Plugin_pass* pass)
{
   pm->add_after_named_pass (pass, new String ("ast"));
}

extern "C" void run_ast (AST::PHP_script* in, Pass_manager* pm, String* option)
{
   printf("%d statement(s) found\n", in->statements->size());
}
```

Save this file to `~/myplugins/count_statements.cpp`. Compile:

```
~/myplugins$ phc_compile_plugin count_statements.cpp
```

And run:

```
./phc --run count_statements.la hello.php
```

Actually..

If you actually did try to run your plugin, you may have found that our plugin isn't quite correct. Consider the following example:

```
<?php
   $x = 5;
   if($x == 5)
      echo "yes";
   else
      echo "no";
?>
```

If you run our plugin on this example, if will report two statements. Why? Well, the first statement is the assignment, and the second is the conditional (the `if` statement). The statements *inside* the `if` statement are not counted, because they are not part of the outer list of statements of the script. In the next tutorial we will see how to fix this.

Writing Stand Alone Applications

If you prefer not to write a plugin but want to modify **phc** itself to derive a new, stand-alone, application, you can add your passes in `src/phc.cpp` in the **phc** source tree instead. This has the effect of "hardcoding" your plugin into **phc** (in versions before 0.1.7, this was the only way to write extensions). However, in the rest of the tutorials we will assume that you are writing your extension as a plugin.

What's Next?

In theory, you now know enough to start implementing your own tools for PHP. Write a new plugin, run the plugin using the `--run` option, and optionally pass in the `--pretty-print` option also to request that **phc** outputs the tree back to PHP syntax after having executed your plugin.

However, you will probably find that modifying the tree, despite being well-defined and easy to understand, is actually rather laborious. It requires a lot of boring boilerplate code. The good news is that **phc** provides sophisticated support for examining and modifying this tree. This is explained in detail in the follow-up tutorials.

Chapter 3. Traversing the Tree

In Chapter 2, we explained that **phc** represents PHP scripts internally as an abstract syntax tree, and that the structure of this tree is determined by the *The Abstract Grammar*. We then showed how to make use of this tree to count the number of statements. However, the plugin we wrote only counted the "top-level" statements. Statements nested inside other statements (for example, statements inside the true or false branch of an `if` statement) were ignored. In this tutorial, we will rectify this problem and write a plugin that counts *all* statements in a script. So, for

```
<?php
   $x = 5;
   if($x == 5)
      echo "yes";
   else
      echo "no";
?>
```

we should report four statements.

Note that all the plugins that we will develop in these tutorials are included in the **phc** distribution. For example, in this tutorial we will be developing two plugins: a difficult solution to the problem and an easy solution to the problem. You can run these plugins by running

```
phc --run plugins/tutorials/count_statements_difficult.la test.php
```

or

```
phc --run plugins/tutorials/count_statements_easy.la test.php
```

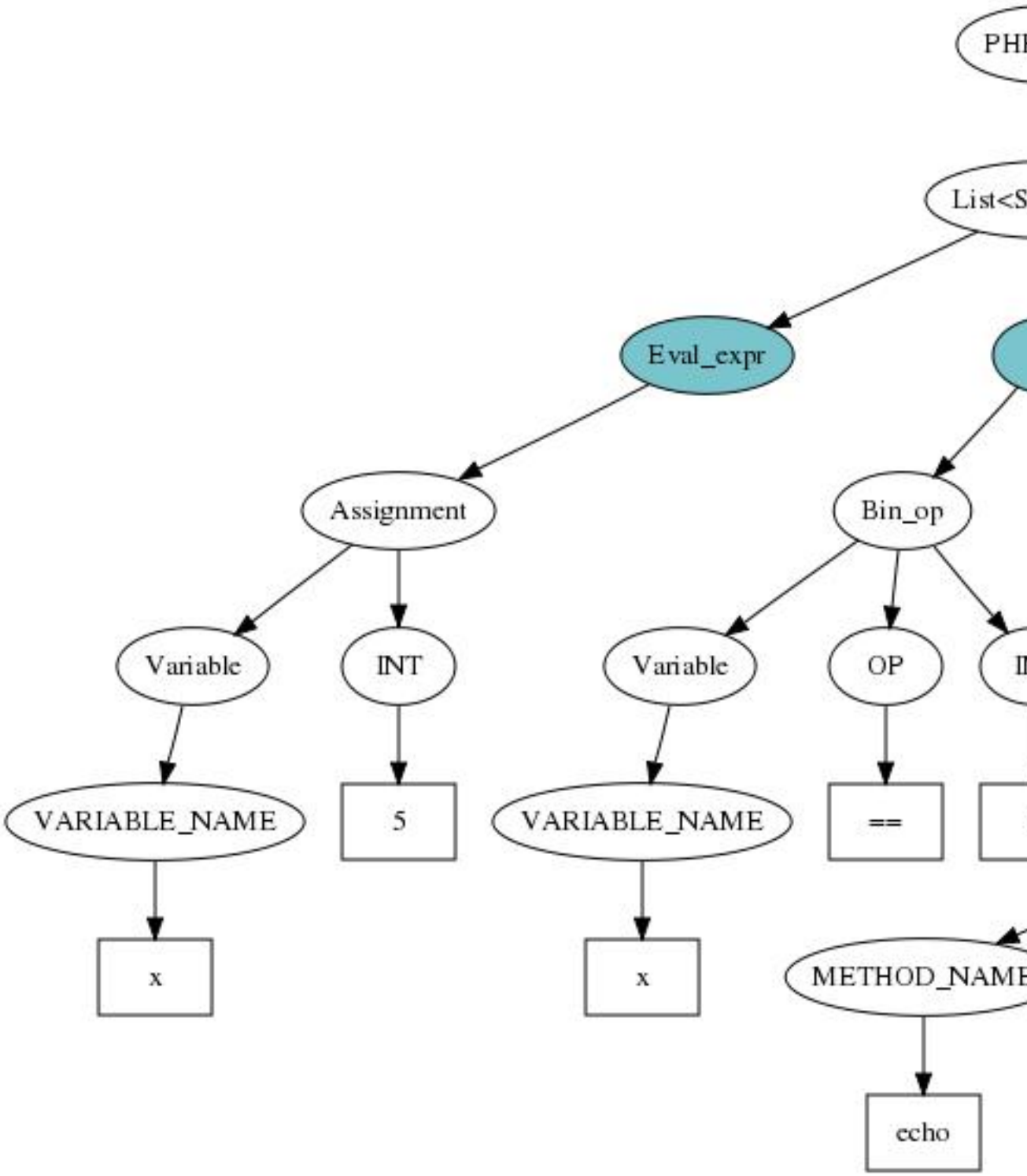
The Grammar (Revisited)

How do we go about counting all statements in a script? Remember that, as far as **phc** is concerned, a PHP script consists of a number of statements, but some of those statements may have other statements nested inside them. Here is part of the **phc** grammar:

```
PHP_script ::= Statement* ;
Statement ::= Eval_expr | If | While | ...
If ::= Expr iftrue:Statement* iffalse:Statement* ;
While ::= Expr Statement* ;
```

The vertical bar (`|`) means "or". So, a statement is either an evaluation of an expression (`eval_expr`), an `if` statement, or a `while` statement, or.. Thus, our running example is represented by the tree in Figure 3-1. The four statements that we are interested in have been highlighted.

Figure 3-1. Abstract syntax tree for the running example



Statements and Expressions

The `Eval_expr` nodes in the tree probably need some explanation. There are many different types of statements in PHP: `if`-statements, `while`-statements, `for`-loops, etc. You can find the full list in the *The Abstract Grammar*. If you do look at the grammar, you will notice in particular that a function call is not actually a statement! Instead, a function call is an *expression*.

The difference between statements and expressions is that a statement *does* something (for example, a `for`-loop repeats a bunch of other statements), but an expression has a *value*. For example, "5" is an expression (with value 5), "1+1" is an expression (with value 2), etc. A function call is also considered an expression. The value of a function call is the value that the function returns.

Now, an `eval_expr` makes a statement from an expression. So, if you want to use an expression where **phc** expects a statement, you have to use the grammar rule

```
Statement ::= ... | Eval_expr ;
Eval_expr ::= Expr ;
```

The Difficult Solution

The following plugin is a partial solution to counting the number of statements in a tree. If you do not understand the code, do not worry! We will look at a much easier solution in a second. If you understand the comments, that is enough.

```
#include <AST.h>
#include <pass_manager/Plugin_pass.h>

int count(AST::Statement_list* in)
{
   // Every item in "in" is a statement
   int num_statements = in->size();

   // But there can also be statements nested inside any
   // of the statements in "in". We consider each one in turn.
   Statement_list::const_iterator i;
   for(i = in->begin(); i != in->end(); i++)
   {
      // Check if the statement is an if-statement
      if(If* if_stmt = dynamic_cast<If*>(*i))
      {
         num_statements += count(if_stmt->iftrue);
         num_statements += count(if_stmt->iffalse);
      }
   }

   return num_statements;
}

extern "C" void load (Pass_manager* pm, Plugin_pass* pass)
{
   pm->add_after_named_pass (pass, new String ("ast"));
```

```
}

extern "C" void run_ast (AST::PHP_script* in, Pass_manager* pm, String* option)
{
   int num_statements = count(in->statements);
   cout << num_statements << " statements found" << endl;
}
```

The overall structure of this plugin should be fairly clear. We count all the statements in the outermost list of statements, and then consider each statement in turn to check if there are any statements nested inside them. This plugin will now report the correct number of statements for our example with the `if` statement. However, if will report an invalid number of statements for examples with other types of statements. For example, it will report only two statements for

```
<?php
   $x = 5;
   while($x--)
   {
      echo $x;
   }
?>
```

Of course, we can fix the plugin by testing for `while` statements. And for `do` statements. And for `foreach`, `switch`, `try`, etc. As mentioned, manually dealing with the syntax tree is a laborious process! Even something as simple as counting the number of statements in a script becomes a large program. Fortunately, there is a much easier solution.

The Easy Solution

Fortunately, **phc** will do all this for you automatically! There is a standard "do-nothing" tree traversal predefined in **phc** in the form of a class called `AST::Visitor` (defined in `AST_visitor.h`). `AST::Visitor` contains methods for each type of node in the tree. **phc** will automatically traverse the abstract syntax tree for you, and call the appropriate method at each node.

In fact, there are *two* methods defined for each type of node. The first method, called `pre_something`, gets called on a node *before* **phc** visits the children of the node. The second method, called `post_something`, gets called on a node *after* **phc** has visited the children of the node. For example, `pre_if` gets called on an `If`, before visiting the statements in the `iftrue` and `iffalse` clauses of the `If`. After all the statements, have been visited, `post_if` gets called.

So, here is an alternative and much easier solution for our problem. This plugin will actually count *all* statements in a script, without having to worry about all the different ways statements can be embedded in other statements. Moreover, even if the internal representation of **phc** changes (for example, if another type of statement gets added), this plugin will still work as-is.

```
#include "AST_visitor.h"
#include <pass_manager/Plugin_pass.h>

class Count_statements : public AST::Visitor
{
```

```
private:
   int num_statements;

public:
   // Set num_statements to zero before we begin
   void pre_php_script(AST::PHP_script* in)
   {
      num_statements = 0;
   }

   // Print the number of function calls when we are done
   void post_php_script(AST::PHP_script* in)
   {
      cout << num_statements << " statements found" << endl;
   }

   // Count the number of function calls
   void post_statement(AST::Statement* in)
   {
      num_statements++;
   }
};

extern "C" void load (Pass_manager* pm, Plugin_pass* pass)
{
   pm->add_after_named_pass (pass, new String ("ast"));
}

extern "C" void run_ast (AST::PHP_script* in, Pass_manager* pm, String* option)
{
Count_statements cfc;
in->visit(&cfc);
}
```

We override a number of methods of the `Visitor` class to implement the functionality we need; the traversal is then taken care of by **phc**.

Pre and Post Methods

We need to be precise about the order in which **phc** calls all these methods. Suppose we have a node `Foo` (say, an if-statement), which *is-a* `Bar` (say, statement), which itself *is-a* `Baz` (say, commented node). Then **phc** calls the visitor methods in the following order:

1. `pre_baz`
2. `pre_bar`
3. `pre_foo`
4. `children_foo` (visit the children of `foo`)
5. `post_foo`

6. `post_bar`
7. `post_baz`

Just to emphasise, if all of the visitor methods listed above are implemented, they will *all* be invoked, in the order listed above. So, implementing a more specific visitor (`pre_foo`) does not inhibit the more general method (`pre_bar`) from being invoked. You can run the `plugins/tutorials/show_traversal_order.la` from the **phc** distribution to see this in action.

> **Note:** (Advanced users) As mentioned above, if you implement `pre_if` (say), the more general methods such as `pre_statement` or `pre_node` will still be invoked. It is possible to override `pre_if_chain` instead; if you override `pre_if_chain`, you are responsible for calling the more general methods manually. If you don't, they will not be called at all.

Chapter 4. Modifying Tree Nodes

Now that we have seen in Chapter 3 how to inspect the tree, in this tutorial we will look at modifying the tree. The task we set ourselves is: replace all calls to `mysql_connect` by calls to `dbx_connect` (dbx (http://pecl.php.net/package/dbx) is a PECL extension to PHP that allows scripts interface with a database independent of the type of the database; this conversion could be part of a larger refactoring process that makes a script written for MySQL work with other databases.)

The tutorial we develop in this tutorial is available as `MySQL2DBX.la` in the **phc** distribution. To see its effect, run **phc** as follows:

```
phc --run plugins/tutorials/MySQL2DBX.la --pretty-print test.php
```

First Attempt

We are interested in all function calls to `mysql_connect`. Let us have a look at the precise definition of a function call according to the *The Abstract Grammar*:

```
Method_invocation ::= Target? Method_name Actual_parameter* ;
Method_name ::= METHOD_NAME | Reflection ;
Actual_parameter ::= is_ref:"&"? Expr ;
Reflection ::= Expr ;
```

(The `target` of a method invocation is the class or object the function gets invoked on, if any. It need not worry us here.) For now, we are only interested in the `Method_name`. The grammar tells us that a `Method_name` is either a `METHOD_NAME` or a node of type `Reflection`. If a symbol is written in CAPITALS in the grammar, that means it refers to a "token", a literal value. In this case, to an actual method name (such as `mysql_connect`). In PHP, it is also possible to call a method whose name is stored in variable; in this case, the function name will be a `Reflection` node (which contains an `Expr`). In this tutorial, we are interested in "normal" method invocations only.

All tokens have an attribute called `value` which corresponds to the value of the token. For most tokens, the type of `value` is a `String*` (consider a `String` to be an STL `string`. However, for some tokens, for example `INT`, `value` has a different type (e.g., `int`). If the token has a non-standard type, it will have method called `get_source_rep`, which returns a `String*` representing the token in the source. For example, the real number `5E-1` would have `value` equal to the (`double`) 0.5, but `get_source_rep` would return (the `String*`) "5E-1".

Thus, we arrive at the following first attempt.

```
#include "AST_visitor.h"
#include <pass_manager/Plugin_pass.h>

using namespace AST;

class MySQL2DBX : public Visitor
{
public:
```

```
   void post_method_invocation(Method_invocation* in)
   {
      if(in->method_name->match(new METHOD_NAME(new String("mysql_connect"))))
      {
         // Modify name
         in->method_name = new METHOD_NAME(new String("dbx_connect"));
      }
   }
};

extern "C" void load (Pass_manager* pm, Plugin_pass* pass)
{
   pm->add_after_named_pass (pass, new String ("ast"));
}

extern "C" void run_ast (PHP_script* in, Pass_manager* pm, String* option)
{
   MySQL2DBX m2d;
   in->visit(&m2d);
}
```

> **Note: phc** uses a garbage collector, so there is never any need to free objects (you never have to call `delete`). This makes programming much easier and less error-prone (smaller chance of bugs).

`match` compares two (sub)trees for deep equality. There is also another function called `deep_equals`, which does nearly the same thing, but there are two important differences. `match` does not take comments, line numbers and other "additional" information into account, whereas `deep_equals` does. The second difference is that `match` supports wildcards; this will be explained in Chapter 5.

Modifying the Parameters

Unfortunately, renaming `mysql_connect` to `dbx_connect` is not sufficient, because the parameters to the two functions differ. According to the PHP manual (http://www.php.net/manual/en/index.php), the signatures for both functions are

```
mysql_connect (server, username, password, new_link, int client_flags)
```

and

```
dbx_connect (module, host, database, username, password, persistent)
```

The `module` parameter to `dbx_connect` should be set to `DBX_MYSQL` to connect to a MySQL database. Then `host` corresponds to `server`, and `username` and `password` have the same purpose too. So, we should insert `DBX_MYSQL` at the front of the list, and insert `NULL` in between `host` and `username` (the `mysql_connect` command does not select a database). The last two parameters to `mysql_connect` do not have an equivalent in `dbx_connect`, so if they are specified, we cannot perform the conversion. The last parameter to `dbx_connect` (`persistent`) is optional, and we will ignore it in this tutorial.

Now, in **phc**, `DBX_MYSQL` is a `Constant`, which has two fields, an optional class name (for class constants) and the name of the constant, of type `CONSTANT_NAME`. `NULL` is represented by `NIL` (to avoid getting confused with the C++ value `NULL`.

We are now ready to write our conversion function:

```
#include "AST_visitor.h"
#include <pass_manager/Plugin_pass.h>

using namespace AST;

class MySQL2DBX : public Visitor
{
public:
   void post_method_invocation(Method_invocation* in)
   {
      Actual_parameter_list*>::iterator pos;
      CONSTANT_NAME* module_name;
      Constant* module_constant;
      Actual_parameter* param;

      if(in->method_name->match(new METHOD_NAME(new String("mysql_connect"))))
      {
         // Check for too many parameters
         if(in->actual_parameters->size() > 3)
         {
            printf("Error: unable to translate call "
               "to mysql_connect on line %d\n", in->get_line_number());
            return;
         }

         // Modify name
         in->method_name = new METHOD_NAME(new String("dbx_connect"));

         // Modify parameters
         module_name = new CONSTANT_NAME(new String("DBX_MYSQL"));
         module_constant = new Constant(NULL, module_name);

         pos = in->actual_parameters->begin();
         param = new Actual_parameter(false, module_constant);
         in->actual_parameters->insert(pos, param); pos++;
         /* Skip host */ pos++;
         param = new Actual_parameter(false, new NIL());
         in->actual_parameters->insert(pos, param);
      }
   }
};

extern "C" void load (Pass_manager* pm, Plugin_pass* pass)
{
   pm->add_after_named_pass (pass, new String ("ast"));
}
```

```
extern "C" void run_ast (PHP_script* in, Pass_manager* pm, String* option)
{
MySQL2DBX m2d;
in->visit(&m2d);
}
```

If we apply this transformation to

```
$link = mysql_connect('host', 'user', 'pass');
```

We get

```
$link = dbx_connect(DBX_MYSQL, "host", NULL, "user", "pass");
```

Refactoring

A quick note on refactoring. Refactoring is the process of modifying existing programs (PHP scripts), usually to work in new projects or in different setups (for example, with a different database engine). Manual refactoring is laborious and error-prone, so tool-support is a must. Although **phc** can be used to refactor PHP code as shown in this tutorial, a dedicated refactoring tool for PHP would be easier to use (though of course less flexible). Such a tool can however be built on top of **phc**.

What's Next?

Chapter 5 explains how you can modify the *structure* of the tree, as well as the tree nodes.

Chapter 5. Restructuring the Tree

Now that we have seen in Chapter 3 how we can traverse the tree, and in Chapter 4 how we can modify individual nodes in the tree, in this tutorial we will look at modifying the structure of the tree itself.

The transform that we will be considering in this tutorial is one that is used in **phc** itself. The transform is called `Remove_concat_null` and can be found in `src/process_ast/Remove_concat_null.h`. The purpose of the transform is to remove string concatenation with the empty string. For example,

```
<?php
   $s = "foo" . "";
?>
```

is translated to

```
<?php
   $s = "foo";
?>
```

The reason that this transform is implemented in **phc** is due to how the **phc** parser deals with in-string syntax. For example, if you write

```
$a = "foo $b bar";
```

the corresponding tree generated by **phc** is

```
$a = "foo " . $b . " bar";
```

In other words, the variables are pulled out of the string, and the various components are then concatenated together. However, taken to its logical conclusion, that means that if you write

```
$a = "foo $b";
```

the parser generates

```
$a = "foo " . $b . "";
```

Obviously, the second concatenation is unnecessary, and the `Remove_concat_null` transform cleans this up. In this tutorial we will explain how this transform can be written.

Introducing the `Tree_transform` API

Concatenation is a binary operator, so we are interested in nodes of type `Bin_op`. If you check the grammar, or, alternatively, `src/generated/AST.h`, you will find that `Bin_op` has three attributes: a `left` and a `right` expression (of type `Expr`) and the operator itself (`OP* op`). Thus, we are interested in nodes of type `Bin_op` whose `op` equals the single dot (for string concatenation).

Based on the previous two tutorials, we might try something like this:

```
class Remove_concat_null : public Visitor
{
public:
   void pre_bin_op(Bin_op* in)
   {
      // Find concat operators
      if(*in->op->value == ".")
      {
         // ...
      }
   }
}
```

The problem is, what are we going to do inside the `if`? Tree visitors can only inspect and modify `*in`; they cannot restructure the tree. In particular, we cannot replace `*in` by a new node. For this purpose, **phc** offers a separate API, the tree *transformation* API. It looks very similar to the tree visitor API, but there are two important differences. First, the `pre` and `post` methods can modify the structure of the tree by returning new nodes. Second, there are no "generic" methods in the tree transform API. So, it is not possible to define a transformation that would replace all statements by something else. (It is not clear how that would be useful, anyway.)

So, we need to write our transformation using the `Tree_transform` API, defined in `AST_transform.h`. Restructuring the class above yields

```
class Remove_concat_null : public Transform
{
public:
   Expr* pre_bin_op(Bin_op* in)
   {
      // Find concat operators
      if(*in->op->value == ".")
      {
         // ...
      }
   }
}
```

The differences between the previous version have been highlighted. We inherit from a different class, and `pre_bin_op` now has a return value, which is the node that will replace `*in`. If you check the default implementation of `pre_bin_op` in `AST_transform.cpp`, you'll find:

```
Expr* Transform::pre_bin_op(Bin_op* in)
{
   return in;
}
```

The `return in;` is very important; as we mentioned before, the return value of `pre_bin_op` will replace `*in` in the tree. Therefore, if we don't want to replace `*in`, or perhaps if we want to replace `*in` only if a particular condition holds, we must return `in`. This will replace `*in` by `in` itself.

The second thing to note is that the return type of `pre_bin_op` is `Expr` instead of `Bin_op`. This means that we can replace a binary operator node by another other expression node. The *Maketea Theory*

explains exactly how the signatures for the `pre` and `post` methods are derived, but in most cases they are what you'd expect. The easiest way to check is to simply look them up in `<AST_transform.h>`.

The Implementation

We wanted to get rid of useless concatenation operators. To be precise, if the binary operator is the concatenation operator, and the left operand is the empty string, we want to replace the node by the right operand; similarly, if the right operand is the empty string, we want to replace the operator by its left operand. Here's the full transform:

```
class Remove_concat_null : public Transform
{
public:
   Expr* post_bin_op(Bin_op* in)
   {
      STRING* empty = new STRING(new String(""));
      Wildcard<Expr>* wildcard = new Wildcard<Expr>;

      // Replace with right operand if left operand is the empty string
      if(in->match(new Bin_op(empty, wildcard, ".")))
         return wildcard->value;

      // Replace with left operand if right operand is the empty string
      if(in->match(new Bin_op(wildcard, empty, ".")))
         return wildcard->value;

      return in;
   }
}
```

We already explained what `match` does in Chapter 4, but we have not yet explained the use of wildcards. If you are using a wildcard (`WILDCARD`) in a pattern passed to `match`, `match` will not take that subtree into account. Thus,

```
if(in->match(new Bin_op(empty, WILDCARD, ".")))
```

can be paraphrased as "is `in` a binary operator with the empty string as the left operand and `"."` as the operator (I don't care about the right operand)?" If the match succeeded, you can find out which expression was matched by the wildcard by accessing `wildcard->value`.

Running Transformations

Recall from the previous two tutorials that visitors are run with a call to `visit`:

```
extern "C" void run_ast (PHP_script* in, Pass_manager* pm, String* option)
{
    SomeVisitor visitor;
    in->visit(&visitor);
```

```
}
```

Likewise, transformations are run with a call to `transform_children`:

```
extern "C" void run_ast (PHP_script* in, Pass_manager* pm, String* option)
{
    SomeTransform transform;
    in->transform_children(&transform);
}
```

We invoke `transform_children` because we should not replace the top-level node in the AST (the `PHP_script` node itself).

A Subtlety

If you don't understand this section right now, don't worry about it; you might find it useful to read it again after having gained some experience with the transformation API.

We have implemented the transform as a *post*-transform rather than a *pre*- transform. Why? Suppose we implemented the transform as a pre-transform. Consider the following PHP expression (bracketed explicitly for emphasis:)

```
("" . $a) . ""
```

The first binary operator we encounter is the second one (get **phc** to print the tree if you don't see why.) So, we apply the transform and replace the operator by its left operand, which happens to be `("" . $a)`. We then continue *and transform the children of the that node*, because that is how the tree transform API is defined. But the *children* of that node are `""` and `$a`. So, that means that the other binary operator itself will never be processed!

There are two solutions to this problem. The first is the one we used above, and use a post-transform instead of a pre-transform. You should try to reason out why this works, but a rule of thumb is that unless there is a good reason to use a pre-transform, it's safer to use the post-transform, because in the post-transform the children of the node have already been transformed, so that you are looking at the "final" version of the node.

The second solution is to use a pre-transform, but explicitly tell **phc** to transform the new node in turn. This is the less elegant solution, but sometimes this is the only solution that will work (see for example the `Token_conversion` transform in the `src/process_ast/Token_conversion.cpp`). To do this, you would replace

```
return in->right;
```

by

```
return in->right->pre_transform(this);
```

What's Next?

The next tutorial in this series, *Using State*, introduces a very important notion in transforms: the use of *state*.

Chapter 6. Using State

This tutorial explains an advanced feature of pattern matching, and shows an important technique in writing tree transforms: the use of state. Suppose we are continuing the refactoring tool that we began in Chapter 4, and suppose that we have replaced all calls to database specific functions by calls to the generic DBX functions. To finish the refactoring, we want to rename any function `foo` in the script to `foo_DB`, if it makes use of the database — this clearly sets functions that use the database apart, which may make the structure of the script clearer.

So, we want to write a transform that renames all functions `foo` to `foo_DB`, if there is one or more call within that function to any `dbx_`*something* function. Here is a simple example:

```
<?php
   function first()
   {
      global $link;
      $error = dbx_error($link);
   }

   function second()
   {
      echo "Do something else";
   }
?>
```

After the transform, we should get

```
<?php
   function first_DB()
   {
      global $link;
      $error = dbx_error($link);
   }

   function second()
   {
      echo "Do something else";
   }
?>
```

The Implementation

Since we have to modify method (function) names, the nodes we are interested in are the nodes of type `Method`. However, how do we know when to modify a particular method? Should we search the method body for function calls to `dbx_`*xxx*? As we saw in Chapter 3, manual searching through the tree is cumbersome; there must be a better solution.

The solution is in fact very easy. At the start of each method, we set a variable `uses_dbx` to `false`. When we process the method, we set `uses_dbx` to `true` when we find a function call to a DBX function. Then at the end of the method, we check `uses_dbx`; if it was set to `true`, we modify the name of the method. This tactic is implement by the following transform (available as `plugins/tutorials/InsertDB.la` in the **phc** distribution). Note the use of `pre_method` and `post_method` to initialise and check `use_dbx`, respectively. (Because we don't need to modify the structure of the tree in this transform, we use the simpler `AST_visitor` API instead of the `AST_transform` API.)

```
class InsertDB : public Visitor
{
private:
   int uses_dbx;

public:
   void pre_method(Method* in)
   {
      uses_dbx = false;
   }

   void post_method(Method* in)
   {
      if(uses_dbx)
         in->signature->method_name->value->append("_DB");
   }

   void post_method_invocation(Method_invocation* in)
   {
      Wildcard<METHOD_NAME>* pattern = new Wildcard<METHOD_NAME>;

      // Check for dbx_
      if(in->method_name->match(pattern) &&
         pattern->value->value->find("dbx_") == 0)
      {
         uses_dbx = true;
      }
   }
};
```

In Chapter 4, we simply wanted to check for a particular function name, and we used `match` to do this:

```
if(in->match(new METHOD_NAME("mysql_connect")))
```

Here, we need to check for method names that start with `dbx_`. We use the STL method `find` to do this, but we cannot call this directly on `in->method_name` because `in->method_name` has type `Method_name` (could either be a `METHOD_NAME` or a `Reflection` node). However, calling `match` on a pattern has the side effect of setting the `value` to point to the node that was matched by the wildcard. So, if the match succeeds, we know that the name of the method must have been a `METHOD_NAME`, and we can access this name by accessing `pattern->value` (`pattern->value->value` is the value field of the `METHOD_NAME` itself, i.e., the actual string that stores the name of the method.)

(Of course, this transform is not complete; renaming methods is not enough, we must also rename the corresponding method invocations. This is left as an exercise for the reader.)

What's Next?

Chapter 7 explains how to change the order in which the children of a node are visited, avoid visiting some children, or how to execute a piece of code in between visiting two children.

Chapter 7. Modifying the Traversal Order

As explained in the previous tutorials (in particular, *Traversing the Tree*), when a `AST_visitor` traverses a tree, it first calls `pre_xxx` for a node of type *xxx*, it then visits all the children of the node, and finally it calls `post_xxx` on the node. For many transforms, this is sufficient — but not for all. Consider the following transform. Suppose we want to add comments to the *true* and *false* branches of an *if*-statement, so that the following example

```
<?php
   if($expr)
   {
      echo "Do something";
   }
   else
   {
      echo "Do something else";
   }
?>
```

is translated to

```
<?php
   if($expr)
   {
      /* TODO: Insert comment */
      echo "Do something";
   }
   else
   {
      /* TODO: Insert comment */
      echo "Do something else";
   }
?>
```

This appears to be a simple transform. One way to do implement it would be to introduce a flag `comment` that is set to `true` when we encounter an `If` (i.e., in `pre_if`). Then in `post_statement` we could check for this flag, and if it is set, we could add the required comment to the statement, and reset the flag to `false`.

However, this will only add a comment to the first statement in the *true* branch (try it!). To add a comment to the first statement in the *false* branch too, we should set the flag to `true` in between visiting the children of the *true* branch and visiting the children of the *false* branch. To be able to do this, we need to modify `children_if`, as explained in the next section.

The Solution

For every AST node type *xxx*, the AST Transform API defines a method called `children_xxx`. This method is responsible for visiting all the children of the node. The default implementation for `If` is:

```
void Visitor::children_if(If* in)
{
    visit_expr(in->expr);
    visit_statement_list(in->iftrue);
    visit_statement_list(in->iffalse);
}
```

(you can find this definition in `AST_visitor.cpp`). If you want to change the order in which the children of a node are visited, entirely avoid visiting some children, or simply execute a piece of code in between two children, this is the method you will need to modify.

Here is the transform that does what we need (available as `plugins/tutorials/Comment_ifs.la`):

```
#include "AST_visitor.h"

class Comment_ifs : public Visitor
{
private:
   bool comment;

public:
   Comment_ifs()
   {
      comment = false;
   }

   void children_if(If* in)
   {
      visit_expr(in->expr);
      comment = true;
      visit_statement_list(in->iftrue);
      comment = true;
      visit_statement_list(in->iffalse);
      comment = false;
   }

   void post_statement(Statement* in)
   {
      if(comment && in->get_comments()->empty())
         in->get_comments()->push_back(new String("/* TODO: Insert comment */"));

      comment = false;
   }
};
```

What's Next?

Chapter 8 explains how to deal with transforms that can replace a single node by multiple new nodes, and shows how to call the **phc** parser and unparser from your plugins.

Chapter 8. Returning Lists

In this tutorial we will develop step-by-step a transform that expands `include` statements. For example, if `b.php` is

```
<?php
   echo "Hello world";
?>
```

and `a.php` is

```
<?php
   include "b.php";
   echo "Goodbye!";
?>
```

Then running the transform on `a.php` yields

```
<?php
   echo "Hello world\n";
   echo "Goodbye\n";
?>
```

The transform we will develop in this tutorial is only a simple implementation of `include`s, and we won't take every feature of `include` into account. However, it can serve as a basis for more full-featured version. The transform we will develop here is available as `plugins/tutorials/Expand_includes.la`.

Deleting Nodes

Our transform should process `include` statements. In the AST, `include`s are represented as method invocations. Thus, we might start like this:

```
class Expand_includes : public Transform
{
public:
   Expr* pre_method_invocation(Method_invocation* in)
   {
      // Process includes
   }
};
```

However, this will not get us very far. The return type of `pre_method_invocation` is an `Expr`. That means that we can replace the method invocation (the `include` statement) only by another, single, expression. But we want to replace it by the contents of the specified file!

Recall from Chapter 3 that to turn an expression into a statement, **phc** inserts an `Eval_expr` in the abstract syntax tree. Thus, if we want to process `include` statements, we could also look at all

`eval_expr` nodes. Assuming for the moment we can make that work, does it get us any further? As a matter of fact, it does! If you check `AST_transform.h`, you will see that the signature for `pre_eval_expr` is

```
void pre_eval_expr(Eval_expr* in, Statement_list* out)
```

This is different from the signatures we have seen so far. For nodes that can be replaced by a number of new nodes, the pre transform and post transform methods will not have a return value in their signature, but have an extra `xxx_List` argument. This list is initialised to be empty before `pre_eval_expr` is invoked, and when `pre_eval_expr` returns, the nodes in this list will replace `*in`. If the list is empty, the node is simply deleted from the tree.

So, we will use the following p lugin as our starting point. Executing this plugin deletes all `Eval_expr` nodes from the tree (try it!).

```
#include "AST_transform.h"

class Expand_includes : public Transform
{
public:
   void pre_eval_expr(Eval_expr* in, Statement_list* out)
   {
   }
};

extern "C" void load (Pass_manager* pm, Plugin_pass* pass)
{
   pm->add_after_named_pass (pass, new String ("ast"));
}

extern "C" void run_ast (PHP_script* in, Pass_manager* pm, String* option)
{
   Expand_includes einc;
   in->transform_children(&einc);
}
```

Using the XML unparser

So, we now want to do something more useful than deleting all `eval_expr` nodes from the tree. The first thing we need to be able to do is distinguish `include` statements from other `eval_expr` nodes. We can use pattern matching (see Chapter 5 and Chapter 6) to do that - but what should we match against? If you are unsure about the structure of the tree, it can be quite useful to use the XML unparser to find out what the tree looks like. We modify the plugin as follows:

```
#include "AST_transform.h"
#include "process_ir/XML_unparser.h"

class Expand_includes : public Transform
{
private:
```

```
   XML_unparser* xml_unparser;

public:
   Expand_includes()
   {
      // Send output to cout, do not print attributes
      xml_unparser = new XML_unparser(cout, false);
   }

public:
   void pre_eval_expr(Eval_expr* in, Statement_list* out)
   {
      in->visit(xml_unparser);
   }
};
```

The XML unparser is implemented using the `Visitor` API, so it can be invoked just like you run any other visitor. There is a similar visitor called `AST_unparser` (in `<process_ast/AST_unparser.h>`) that you can use to print (parts of the) AST to PHP syntax.

When you run this transform on `a.php`, it will print two `eval_expr` nodes (shown in XML syntax), one for the `include` and one for the `echo` . We are interested in the first, the `include`:

```
<AST:Eval_expr>
   <AST:Method_invocation>
      <AST:Target xsi:nil="true" />
      <AST:METHOD_NAME>
         <value>include</value>
      </AST:METHOD_NAME>
      <AST:Actual_parameter_list>
         <AST:Actual_parameter>
            <bool><!-- is_ref -->false</bool>
            <AST:STRING>
               <value>b.php</value>
            </AST:STRING>
         </AST:Actual_parameter>
      </AST:Actual_parameter_list>
   </AST:Method_invocation>
</AST:Eval_expr>
```

This tells us that the `include` statement is an `Eval_expr` node (that was obvious from the fact that we implemented `pre_eval_expr`). The `Eval_expr` contains a `Method_invocation` (we knew that too, although of course a node of type `Eval_expr` can contain any type of expression). The method invocation has target `NULL` (it is not invoked on an object or a class), method name "`include`", and a single parameter in the parameter list that contains the name of the file we are interested in. We can construct a pattern that matches this tree exactly:

```
class Expand_includes : public Transform
{
private:
   Wildcard<STRING>* filename;
   Method_invocation* pattern;
```

```
public:
   Expand_includes()
   {
      filename = new Wildcard<STRING>;
      pattern =
         new Method_invocation(
            NULL,
            new METHOD_NAME(new String("include")),
            new List<Actual_parameter*>(
               new Actual_parameter(false, filename)
            )
         );
   }

public:
   void pre_eval_expr(Eval_expr* in, List<Statement*>* out)
   {
      // Check for calls to include
      if(in->expr->match(pattern))
      {
         // Matched! Try to parse the file
      }
      else
      {
         // No match; leave untouched
         out->push_back(in);
      }
   }
};
```

Note how the construction of the pattern follows the structure of the tree as output by the XML unparser exactly. The only difference is that we leave the actual filename a wildcard; obviously, we want to be able to match against any `include`, not just `include("a.php")`. Running this transform should remove the `include` from the file, but leave the other statements untouched (note that we need to `push_back in` to `out` to make sure a statement does not get deleted).

The Full Transform

We are nearly done! All that's left is to parse the file (we can use the "`filename`" wildcard to find out which file we need to include) and insert all statements into the parsed file at the point of the include. Parsing PHP is hard, but of course **phc** comes with a PHP parser. To use this parser, include the `<parsing/parse.h>` header and call "`parse`". Here then is the full transform:

```
#include "AST_transform.h"
#include "parsing/parse.h"
#include "process_ir/XML_unparser.h"

class Expand_includes : public Transform
{
private:
```

```
   XML_unparser* xml_unparser;
   Wildcard<STRING>* filename;
   Method_invocation* pattern;

public:
   Expand_includes()
   {
      xml_unparser = new XML_unparser(cout, false);

      filename = new Wildcard<STRING>;
      pattern =
         new Method_invocation(
            NULL,
            new METHOD_NAME(new String("include")),
            new List<Actual_parameter*>(
               new Actual_parameter(false, filename)
            )
         );
   }

public:
   void pre_eval_expr(Eval_expr* in, List<Statement*>* out)
   {
      // in->visit(xml_unparser);

      // Check for calls to include
      if(in->expr->match(pattern))
      {
         // Matched! Try to parse the file
         PHP_script* php_script = parse(filename->value->value, NULL, false);
         if(php_script == NULL)
         {
            cerr
            << "Could not parse file " << *filename->value->value
            << " on line " << in->get_line_number() << endl;
            exit(-1);
         }

         // Replace the include by the statements in the parsed file
         out->push_back_all(php_script->statements);
      }
      else
      {
         // No match; leave untouched
         out->push_back(in);
      }
   }
};

extern "C" void load (Pass_manager* pm, Plugin_pass* pass)
{
   pm->add_after_named_pass (pass, new String ("ast"));
}
```

```
extern "C" void run_ast (PHP_script* in, Pass_manager* pm, String* option)
{
   Expand_includes einc;
   in->transform_children(&einc);
}
```

Exercise. One problem with the plugin we have developed is that if the file we are including in turn has `include` statements, they will not be processed. Modify the plugin to invoke the transform on the list of statements from the parsed file, taking care to deal with infinite loops (if the first file includes the second, and the second the first).

What's Next?

This is the last tutorial in this series on using the `AST_visitor` and `AST_transform` classes. Of course, the only way to really learn this stuff is to try it out for yourself. Hopefully, the tutorials will help you do so. The following sources should also be useful:

- The *The Abstract Grammar* (and the *Maketea Theory*)
- The explanation of how PHP gets represented in the abstract syntax as detailed in *Representing PHP*
- The definition of the C++ classes for the AST nodes in `src/generated/AST.h`
- The definition of the `AST_visitor` and `AST_transform` classes in `src/generated/AST_visitor.h` and `src/generated/AST_transform.h>` respectively

And of course, we are more than happy to answer any other questions you might still have. Just send an email to the mailing list (http://www.phpcompiler.org/mailinglist.html) and we'll do our best to answer you as quickly as possible! Happy coding!

II. Reference

Chapter 9. The Abstract Grammar

This is the full and authoritative definition of the **phc** abstract grammar for PHP in `maketea` format (this can also be found in `src/generated_src/ast.tea` in the distribution). For a description of the structure of the grammar, and how it converts to C++ code, refer to the Chapter 13.

Overall Structure

```
PHP_script ::= Statement* ;

Class_def ::=
   Class_mod CLASS_NAME extends:CLASS_NAME?
   implements:INTERFACE_NAME* Member* ;
Class_mod ::= "abstract"? "final"? ;

Interface_def ::= INTERFACE_NAME extends:INTERFACE_NAME* Member* ;

Member ::= Method | Attribute ;

Method ::= Signature Statement*? ;
Signature ::= Method_mod is_ref:"&"? METHOD_NAME Formal_parameter* ;
Method_mod ::= "public"? "protected"? "private"? "static"? "abstract"? "final"? ;
Formal_parameter ::= Type is_ref:"&"?
   var:Name_with_default ;
Formal_parameter ::= Type is_ref:"&"?
   var:Name_with_default ;
Type ::= CLASS_NAME? ;
Name_with_default ::= VARIABLE_NAME Expr? ;

Attribute ::= Attr_mod vars:Name_with_default* ;
Attr_mod ::= "public"? "protected"? "private"? "static"? "const"?  ;
```

Statements

```
Statement ::=
     Class_def | Interface_def | Method
   | Return | Static_declaration | Global
   | Try | Throw | Eval_expr
   | If | While | Do | For | Foreach
   | Switch | Break | Continue
   | Declare | Nop
;

If ::= Expr iftrue:Statement* iffalse:Statement* ;
```

```
While ::= Expr Statement* ;
Do ::= Statement* Expr ;
For ::= init:Expr? cond:Expr? incr:Expr? Statement* ;
Foreach ::= Expr key:Variable? is_ref:"&"?
   val:Variable Statement* ;

Switch ::= Expr Switch_case* ;
Switch_case ::= Expr? Statement* ;
Break ::= Expr? ;
Continue ::= Expr? ;
Return ::= Expr? ;

Static_declaration ::= vars:Name_with_default* ;
Global ::= Variable_name* ;

Declare ::= Directive+ Statement* ;
Directive ::= DIRECTIVE_NAME Expr ;

Try ::= Statement* catches:Catch* ;
Catch ::= CLASS_NAME VARIABLE_NAME Statement* ;
Throw ::= Expr ;

Eval_expr ::= Expr ;

Nop ::= ;
```

Expressions

```
Expr ::=
     Assignment
| Cast | Unary_op | Bin_op
| Constant | Instanceof
| Variable | Pre_op
| Method_invocation | New
| Literal
| Op_assignment | List_assignment
| Post_op | Array | Conditional_expr | Ignore_errors
;

Literal ::= INT<long> | REAL<double> | STRING<String*> | BOOL<bool> | NIL<> ;

Assignment ::= Variable is_ref:"&"? Expr ;
Op_assignment ::= Variable OP Expr ;

List_assignment ::= List_element?* Expr ;
List_element ::= Variable | Nested_list_elements ;
Nested_list_elements ::= List_element?* ;

Cast ::= CAST Expr ;
Unary_op ::= OP Expr ;
```

```
Bin_op ::= left:Expr OP right:Expr ;

Conditional_expr ::=
   cond:Expr iftrue:Expr iffalse:Expr ;
Ignore_errors ::= Expr ;

Constant ::= CLASS_NAME? CONSTANT_NAME ;

Instanceof ::= Expr Class_name ;

Variable ::= Target? Variable_name array_indices:Expr?* ;
Variable_name ::= VARIABLE_NAME | Reflection ;
Reflection ::= Expr ;

Target ::= Expr | CLASS_NAME ;

Pre_op ::= OP Variable ;
Post_op ::= Variable OP ;

Array ::= Array_elem* ;
Array_elem ::= key:Expr? is_ref:"&"? val:Expr ;

Method_invocation ::= Target? Method_name Actual_parameter* ;
Method_name ::= METHOD_NAME | Reflection ;

Actual_parameter ::= is_ref:"&"? Expr ;

New ::= Class_name Actual_parameter* ;
Class_name ::= CLASS_NAME | Reflection ;
```

Additional Structure

```
Commented_node ::=
  Member | Statement | Interface_def | Class_def | Switch_case | Catch
;

Identifier ::=
  INTERFACE_NAME | CLASS_NAME | METHOD_NAME | VARIABLE_NAME
  | CAST | OP | CONSTANT_NAME
  | DIRECTIVE_NAME
;

Source_rep ::= Identifier | Literal ;
```

Mix-in Code

The code generated based on the grammar listed above can be extended by "mix-in" code, which adds fields or methods to the class structure generated by **phc**. For a full listing of the mix-in code, see `src/generated_src/ast.tea` in the **phc** distribution.

Chapter 10. Representing PHP

Most PHP constructs can immediately be represented in terms of the **phc** grammar (Chapter 9). There are a few constructs that present some difficulties. This document describes how these difficulties are resolved, and it explains some of the more difficult rules in the grammar.

Variables

The grammar rule for variables reads

```
variable ::= target? variable_name array_indices:(expr?)* string_index:expr?
variable_name ::= VARIABLE_NAME | reflection
```

This is probably one of the more difficult rules in the grammar, so it is worth explaining in a bit more detail. The following table describe each element of the first rule in detail.

`Target?`

Just like function calls, variables can have a target, and just as for function calls, this target can be an expression (for an object, e.g., `$x->y`) or a class name (for a static class attribute, e.g. `FOO::$y`). As in function calls, in variables the target is optional (indicated by the question mark). If no target is specified, the variable refers to a *local* variable in a method.

`Variable_name`

Again, as for function calls, the name of the variable may be a literal `VARIABLE_NAME` (`$x`), or be given by an expression (which is wrapped up in an `Reflection` node). The latter possibility is referred to as "variable-variables" in the PHP manual. For example, `$$x` is the variable whose name is currently stored in (another) variable called `$x`.

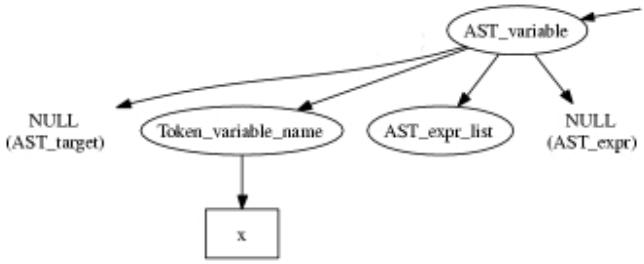
`array_indices:(Expr?)*`

A variable may have one or more array indices, for example `$x[3][5]`. The strange construct `(Expr?)*` means: a list of (`*`) optional (`?`) expressions. For example, `$x[4][]` is a list of two expressions, but the second expression is not given. In PHP, this means "use the next available index". String- and array-indexing (`$x{3}`) are equivalent in PHP, so string-indexing is also represented by `array_indices`.

We illustrate the various possibilities using diagrams.

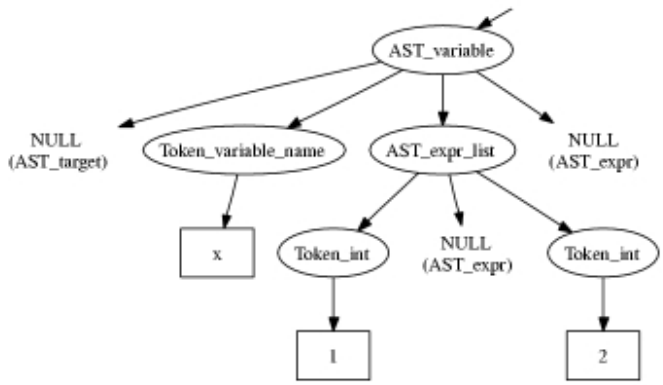
> **Warning**
>
> These diagrams use old names for AST nodes. Where you see `AST_variable`, it is now called `Variable` (it uses the AST namespace). `Token_variable_name` is called `VARIABLE_NAME` and `Token_int` is called `INT`.
>
> It is possible the structure of some nodes have changed slightly since this was written.

The simple case: `$x`



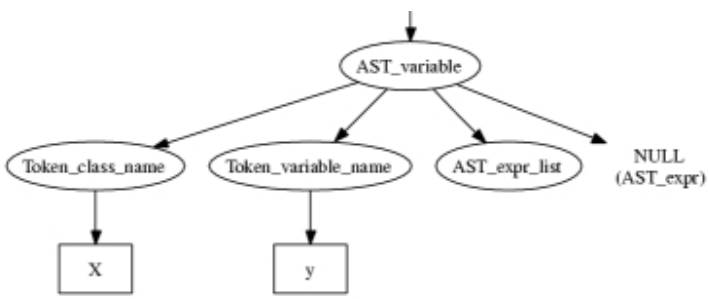
The name of the variable is `x`, not `$x`.

Using array indices: `$x[1][][2]`



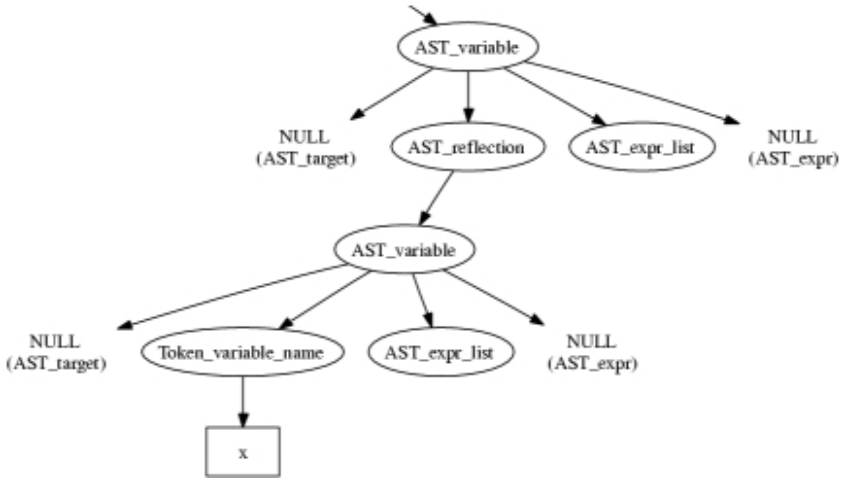
(The empty array index means “next available” in PHP).

Class constants: `X::$y`



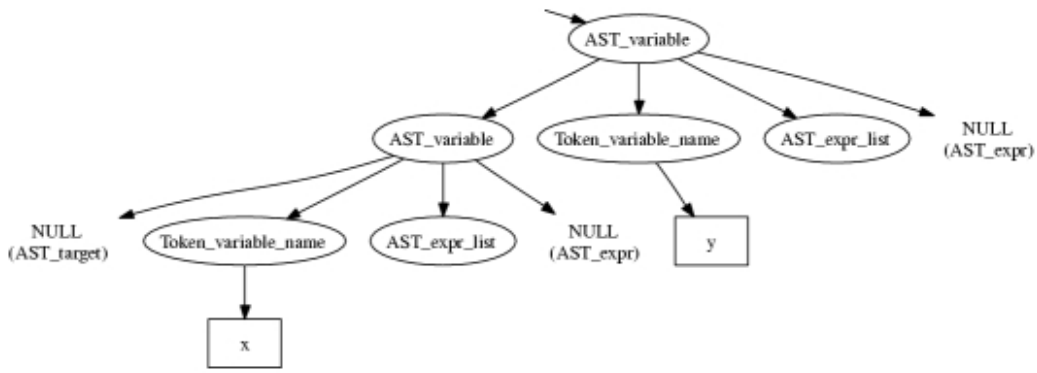
Again, the variable name is `y`, not `$y`. The fact that you must write `$x->y` but `X::$y` in PHP disappears in the abstract syntax.

Variable variables: `$$x`



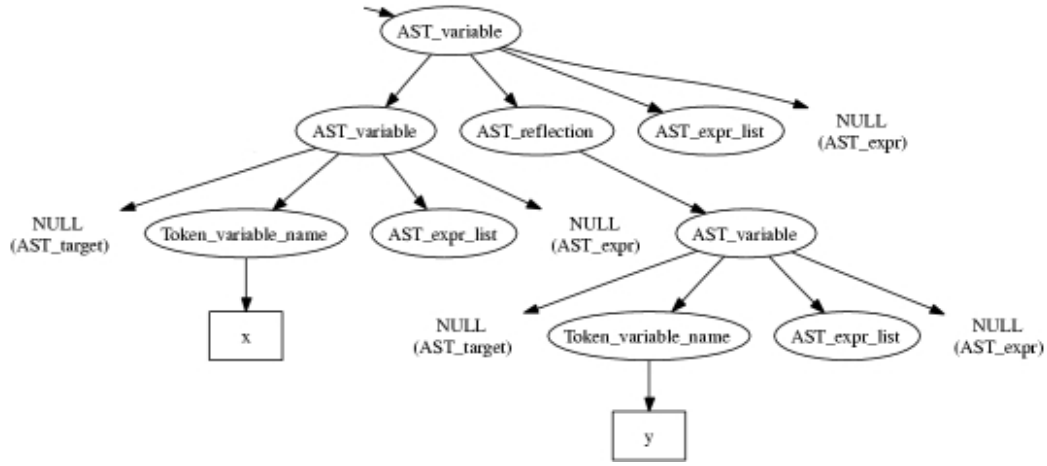
Note how the name of the variable (second component) is now given by another variable.

Object attributes: `$x->y`



The target is now given by a variable.

Variable object attributes: `$x->$y`



Both the target and the variable name are given by (other) variables.

Comments

A number of nodes in the AST are dedicated "commented nodes". Their corresponding C++ classes inherit from `Commented_node`, which introduces a `String_list*` attribute called `comments`. The commented nodes are class members (`Member`), statements (`Statement`), interface and class definitions (`Interface_def`, `Class_def`), switch cases (`Switch_case`) and catches (`Catch`).

When the parser encounters a comment in the input, it attaches it either to the previous node in the AST, or to the next, according to a variable `attach_to_previous`. This variable is set as follows:

- It is reset to `false` at the start of each line
- It is set to `true` after seeing a semicolon, or either of the keywords `class` or `function`

Thus, in

```
foo();
// Comment
bar();
```

the comment gets attached to `bar();` (to be precise, to the corresponding `Eval_expr` node; the function call itself is an expression and **phc** does not associate comments with expressions), but in

```
foo(); // Comment
bar();
```

the comment gets attached to `foo();` instead. The same applies to multiple comments:

```
foo(); /* A */ /* B */
```

```
// C
// D
bar();
```

In this snippet, `A` and `B` get attached to `foo();`, but `C` and `D` get attached to `bar();`. Also, in the following snippet,

```
// Comment
echo /* one */ 1 + /* two */ 2;
```

all comments get attached to the same node. This should work most of the time, if not all the time. In particular, it should never loose any comments. If something goes wrong with comments, please send (http://www.phpcompiler.org/contact.html) us a sample program that shows where it goes wrong. Note that whitespace in multi-line comments gets dealt with in a less than satisfactory way; see *Limitations* for details for details.

String parsing

Double quoted strings and those written using the HEREDOC syntax are treated specially by PHP: it parses variables used inside these strings and automatically expands them with their value. **phc** handles both the simple and complex syntax defined by PHP for variables in strings. We transform a string like

```
"Total cost is: $total (includes shipping of $shipping)"
```

into:

```
"Total cost is: " . $total . " (includes shipping of " . $shipping . ")"
```

which is represented in the **phc** abstract syntax tree by a number of strings and expressions concatenated together. Thus, as a programmer you don't need to do anything special to process variables inside strings. Any code you write for processing variables will also appropriately handle variables inside strings. (Note that as of version 0.2.0, interpolated strings are correctly unparsed by **phc**.)

`elseif`

The abstract grammar does not have a construct for `elseif`. The following PHP code

```
<?php
   if($x)
      c1();
   elseif($y)
      c2();
   else
      c3();
?>
```

gets interpreted as

```
<?php
   if($x)
      c1();
   else
   {
      if($y)
         c2();
      else
         c3();
   }
?>
```

The higher the number of `elseif`s, the greater the level of nesting. This transformation is "hidden" by the unparser.

Miscellaneous Other Changes

- Fragments of inline HTML become arguments to a function call to `echo`.
- The keywords `use`, `require`, `require_once`, `include`, `include_once`, `isset` and `empty` all get translated into a function call to a function with the same name as the keyword.
- `exit` also becomes a call to the function `exit`; `exit;` and `exit();` are interpreted as `exit(0)`

Comparison to the PHP grammar

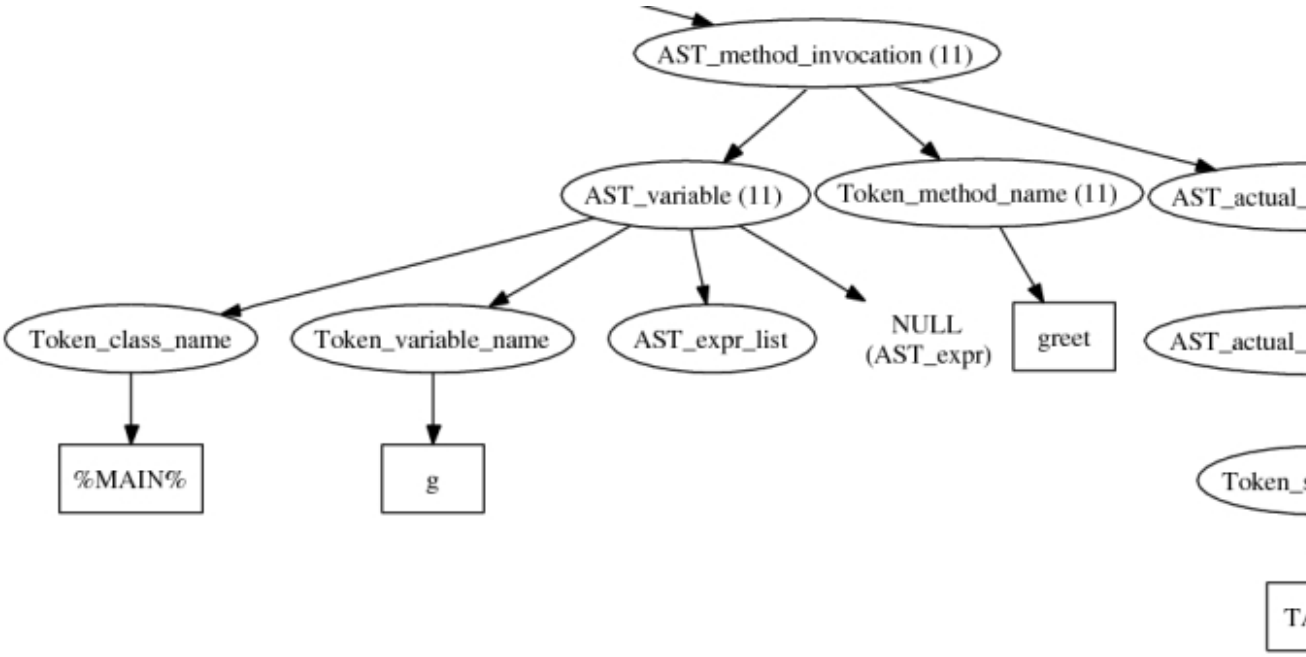
Finally, the **phc** grammar is much simpler than the official grammar, and as a consequence more general. The class of programs that are valid according to the abstract grammar is larger than the class of programs actually accepted by the PHP parser. In other words, it is possible to represent a program in the abstract syntax that does not have a valid PHP equivalent. The advantage of our grammar is that is much, *much* easier to work with.

To compare, consider the tree for

```
$g->greet("TACS");
```

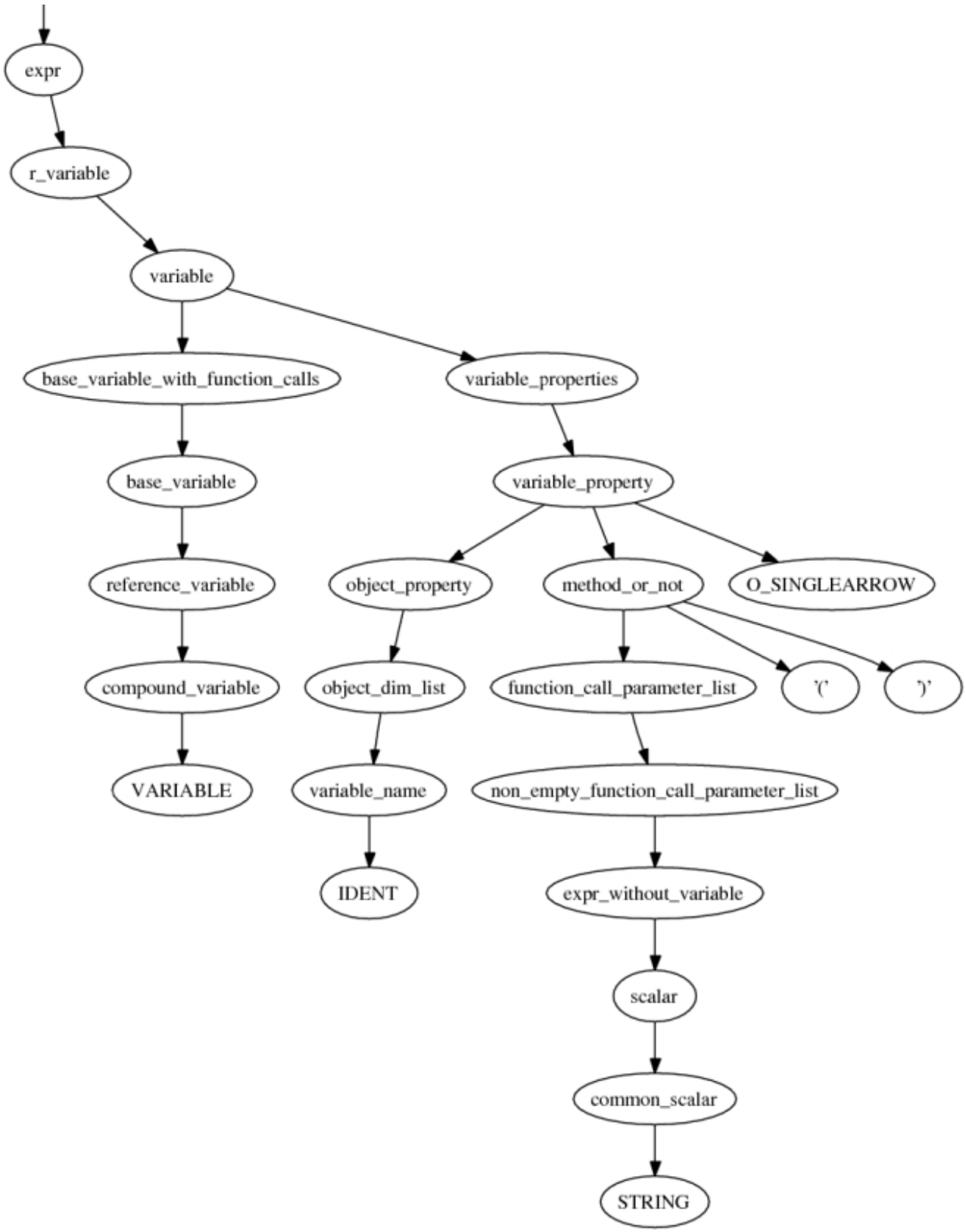
Using the **phc** abstract syntax, this looks like the tree shown in figure Figure 10-1.

Figure 10-1. Function call in the AST



However, in the official PHP grammar, the tree would look like the tree shown in figure Figure 10-2.

Figure 10-2. Function call as represented by PHP



Not only is the number of concepts used in the tree much larger (`base_variable_with_function_calls`, `reference_variable`, `variable_property`, etc. etc.), the concepts used in the **phc** tree map directly to constructs in the PHP language; that does not hold true for the PHP tree. Moreover, the fact that this expression is a method invocation (function call) is immediately obvious from the root of the expression in the **phc** tree; the root of the PHP tree says that the expression is a variable, and only deeper down the tree does it become apparent that the expression is in fact a function call.

Chapter 11. Limitations

This document describes the known limitations of the current **phc** implementation. These limitations are things that we are aware of but that are not high on our priority list of things to deal with at the moment. However, if any of them are bothering you, let us know (http://www.phpcompiler.org/mailinglist.html) and we might look into it.

Comments

Representing PHP explains how we deal with comments. Most comments in a PHP script should get attached to the right token in the tree, and no comments should ever be lost. If that is not true, please send us a sample program that demonstrates where it breaks. There are a few problems that we are aware of, and there are probably others too.

Dealing with comments in a completely satisfactory way is a difficult task! The first problem with our method of dealing with comments is how we deal with whitespace in multi-line comments. Consider the following example.

```
<?php
   /*
    * Some comment with
    * multiple lines
    */
   foo();
?>
```

The problem is that the whitespace at the start of each line is included in the comment. This means that when the unparser outputs the comment, it outputs something like

```
<?php
   /*
    * Some comment with
       * multiple lines
       */
   foo();
?>
```

It is unclear how to solve this problem nicely. Suggestions are welcome :-)

Second, it is not currently possible to associate a comment with the `else`-clause of an `if`-statement. Thus, in

```
<?php
   // Comment 1
   if($c)
   {
      foo();
   }
```

```
   // Comment 2
   else
   {
      bar();
   }
?>
```

Comment 2 will be associated with the call to `bar` (but Comment 1 will be associated with the `if`-statement itself). A similar problem occurs with comments for `elseif` statements.

Finally, if a scope ends on a comment, that comment will be associated with the wrong node. For example, in

```
<?php
   if($c)
   {
      echo "Hi";
   }
   else
   {
      // Do nothing
   }

   echo "World";
?>
```

the comment will be associated with the `echo "World"` statement. A similar problem occurs when a script ends on a comment; that comment will not be lost, but will be associated with the last node in the script.

Numbers

PHP accepts invalid octal numbers such as `01090`; the "incorrect tail" is silently ignored (so, this number should evaluate to 8 decimal). The **phc** lexical analyser will generate an "invalid token" instead which will result in a syntax error.

Scopes

We incorrectly represent

```
<?php
{
function x () {}
}
?>
```

as

```
<?php
```

```
function x () {}
?>
```

In the former, `x` is only declared when its declaration is executed. In the latter, it is declared as soon as the program starts.

Other issues

There are quite a number of minor bugs and issues with **phc**, that we are aware of. Our bug tracker is available at our project site (http://code.google.com/p/phc/issues/list). We are looking for contributors to help us fix many of these bugs. Please see our contributors page (http://phpcompiler.org/contribute.html), if you're interested in helping out.

Chapter 12. Overview of the AST classes and transformation API

This document explains the code for the AST classes, tree visitor API and tree transformation API. All this code is generated by a tool called `maketea`. It does not explain how this code is derived from the **phc** grammar; some of the details of this process are explained in *Maketea Theory*.

The AST classes

There are two main kinds of AST classes: classes that correspond to non-terminals in the grammar, and classes that correspond to terminals in the grammar. Non-terminal classes contain an upper-case first letter. Terminals, or tokens, are entirely uppercase. Examples are `While`, `Expr`, `METHOD_NAME` and `INT`.

The main difference is that token classes have one additional field (and sometimes two). Every token class gets an attribute called `value`. The type of this attribute depends on the token; for most tokens it is `String*` (this is the default); however, if the grammar explicitely specifies a type for the value (in angular brackets, for example `REAL<double>`), this overrides the default.

In addition, all the tokens classes have a method called `get_value_as_string()` and a method `get_source_rep` when applicable. This is useful for programs that operate on general `Identifier` objects (such as `METHOD_NAME` or `CLASS_NAME`) or `Literal` (such as `REAL` or `INT`). Note that the value returned by `get_value_as_string()` and `get_source_rep()` may be different; for example, `get_source_rep()` might return `0.5E-1`, while `get_value_as_string()` might return `0.5`.

All (non-terminal and terminal) then provide the following methods for deep equality, pattern matching, cloning, calling a tree visitor and calling a tree transformer. These methods are explained separately in sections below.

Deep Equality

Deep equality is implemented by `bool deep_equals(Object* other)`. It takes into account the entire tree structure generated by `maketea`, including any fields that are specified in the code in the grammar (see the Section called *Mix-in Code* in Chapter 9). Thus, `deep_equals` also compares line numbers, comments, etc.

Cloning

Cloning is implemented by `deep_clone`. Cloning makes a (deep) copy of a tree, so the set of all pointers in the new tree is completely distinct from the set of pointers in the old tree. The only exception

to this rule is that cloning the `WILDCARD` objects (see pattern matching, below) returns the `WILDCARD` object itself.

Pattern Matching

Pattern matching is implemented by `bool match(Object* pattern)`. Pattern matching differs from deep equality in two ways. First, it does not take into account any fields added by the mixin code; for example, it does not compare line numbers or comments.

Second, `match` supports the use of wildcards. `Maketea` generates a special class called `Wildcard`. You should never instantiate this class directly; in `<AST.h>`, you will find the following declaration:

```
extern Wildcard* WILDCARD;
```

This `WILDCARD` is the sole instance of `Wildcard`. When `match` encounters a reference to this object in a pattern, it does two things: it skips that field in the comparison (so it acts as a "don't care"), and it replaces the value of the field in the pattern by the value in the tree. For example, in the body of the `if` in

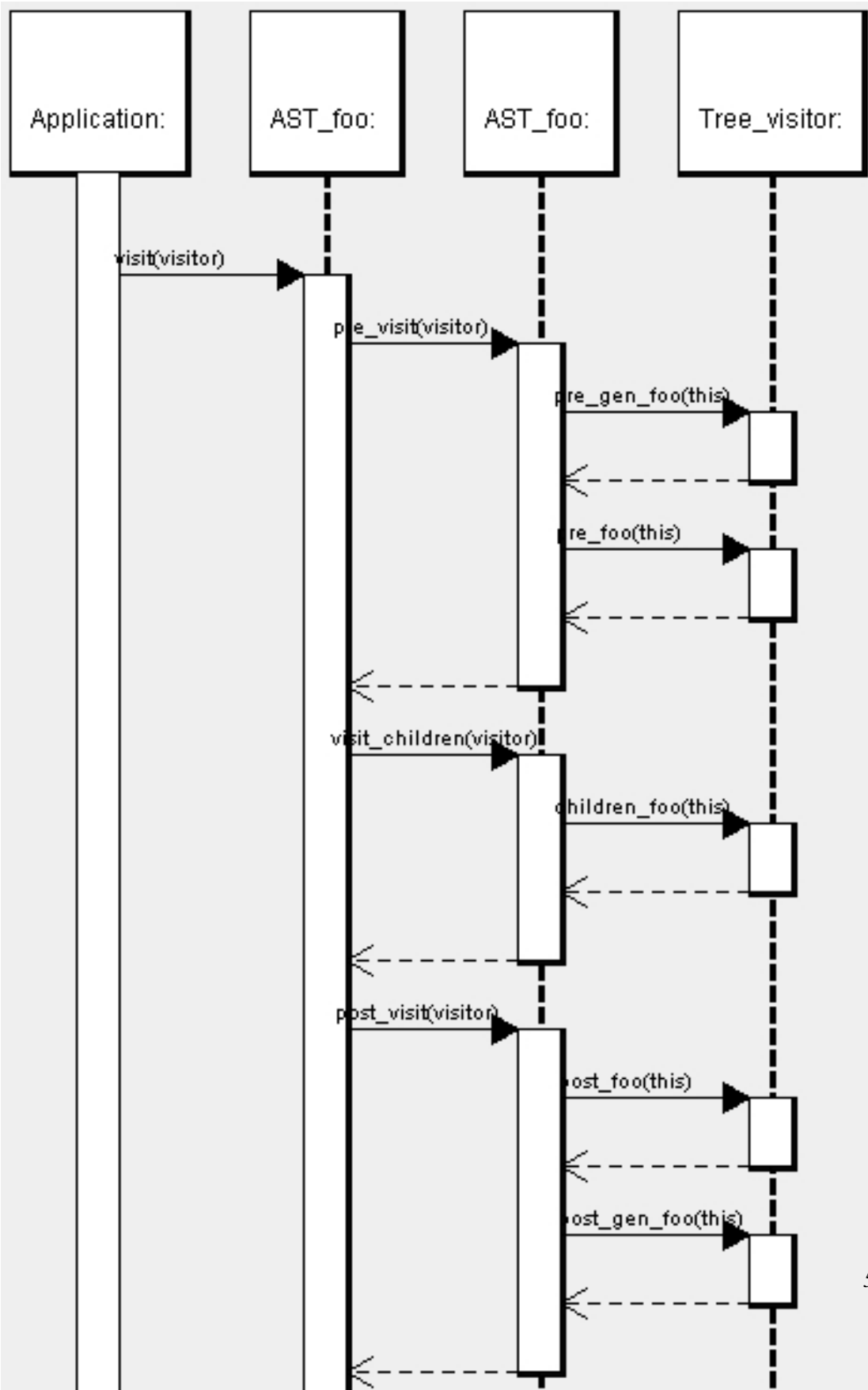
```
CLASS_NAME* name = new CLASS_NAME(new String("SOME_CLASS"));
CLASS_NAME* pattern = new CLASS_NAME(WILDCARD);

if(name->match(pattern))
{
   // ...
}
```

`pattern->value` will be set to the corresponding value in `name`. Tutorials *Restructuring the Tree* and *Using State* include examples of the use of wildcards.

Calling any methods on the `WILDCARD` object other than `deep_clone` will lead to a runtime error.

The Visitor API

Figure 12-1. Sequence Diagram for the Visitor API



Every AST class provides four methods to support the visitor API: `void visit(AST::Visitor*)`, `void pre_visit(AST::Visitor*)`, `void visit_children(AST::Visitor*)` and `void post_visit(AST::Visitor*)`. The implementation of each of these methods is very simple.

`visit` simply calls `pre_visit`, `visit_children` and `post_visit` in order. It could have been implemented once and for all in the `Node` class (but is not, for no particular reason).

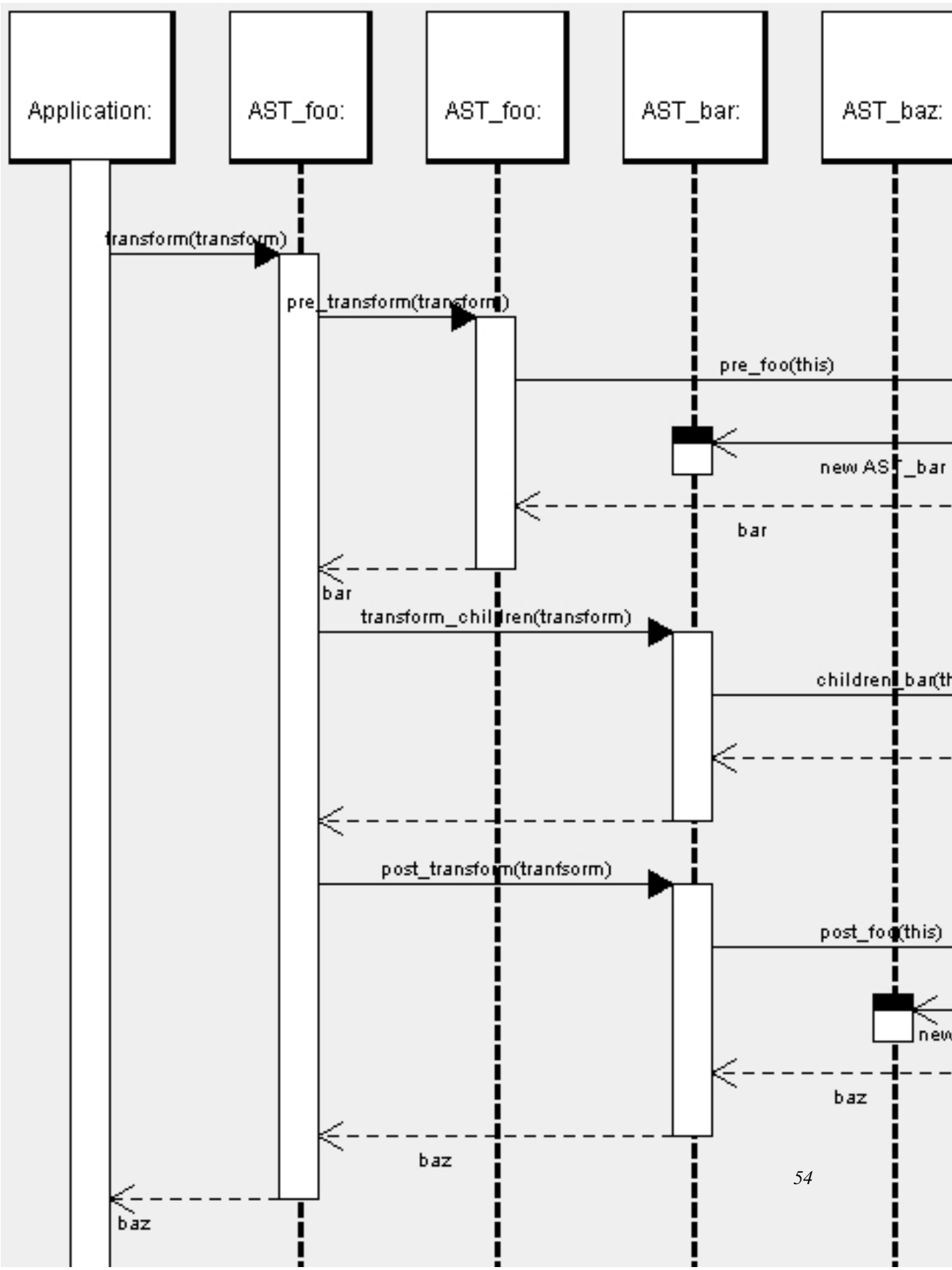
For a node `x0`, which inherits from `x1`, which inherits from `x2`, which in turn inherits from `x3`, etc., `x0::pre_visit` calls `pre_x3`, `pre_x2`, `pre_x1` and `pre_x0`, in that order, on the tree visitor object, passing itself as an argument. If `x0` inherits from multiple classes, all of the appropriate visitor methods will be invoked. However, if `x0` inherits from both `x1a` and `x1b`, the programmer should not rely on the relative order of `pre_x1a` and `pre_x1b`.

`x0::visit_children` simply calls `children_x0`.

`x0::post_visit` will call `post_x0`, `post_x1`, etc. Again, if `x0` inherits from both `x1a` and `x1b`, the programmer should not rely on the relative order of `post_x1a` and `post_x1b`. The only guarantee made by `maketea` is that the order of the pre-methods will be the exact reverse of the order of the post-methods.

The Transform API

Figure 12-2. Sequence Diagram for the Transform API



Every AST class `AST_foo`, which inherits from `AST_gen_foo` provides four methods to support the tree visitor API: `AST_gen_foo* transform(AST::Transform*)`, `AST_gen_foo* pre_transform(AST::Transform*)`, `void transform_children(AST::Transform*)` and `AST_gen_foo* post_transform(AST::Transform*)`. It is not entirely as straightforward as this; if `AST_foo` inherits from more than one class, the return type would probably be `AST_foo`; in some cases, `transform` might return a `AST_foo_list` instead. See the section *Context Resolution* in the grammar formalism for details; here we consider the programmer's perspective only. The exact signatures for a particular class can always be found in `<AST.h>`.

As with the visitor API, `transform` calls `pre_transform`, `transform_children` and `post_transform`. However, while `transform` calls `pre_transform` on itself, it calls `transform_children` and `post_transform` on the node returned by `pre_transform`. If `pre_transform` returns a vector, `transform` calls `transform_children` and `post_transform` on every element in that vector, assembling all the results.

`pre_transform` and `post_transform` simply call the appropriate method in the `AST::Transform` object. However, if `pre_transform` (or `post_transform`) returns a list of nodes, the corresponding method in the tree transform object will expect two arguments: the node to be transformed, and an empty list of nodes that will be the return value of `pre_transform`. In that case, `pre_transform` will first create a new empty list, pass that in as the second argument to the corresponding method in the tree transform object, and then return that list.

`transform_children` just calls the corresponding method in the tree transform object.

Chapter 13. `Maketea` Theory

Introduction

`maketea` is available separately (http://www.maketea.org) to **phc**. Based on a grammar definition of a language, it generates a C++ hierarchy for the corresponding abstract syntax tree, a tree transformation and visitor API, and deep cloning, deep equality and pattern matching on the AST. In this document we describe the grammar formalism used by **phc**, how a C++ class structure is derived from such a grammar, and explains how the tree transformation API is generated. The generated code itself is explained in *Overview of the AST classes and transformation API*.

The Grammar Formalism

The style of grammar formalism used by `maketea` is sometimes referred to as an "object oriented" context free grammar. It facilitates a trivial and reliable mapping between the grammar (Chapter 9, and the actual (C++) abstract syntax tree (AST) that is generated by the **phc** parser.

We make a distinction between three types of symbols: *non-terminal* symbols, *terminal symbols* and *markers*. Non-terminal symbols have the same function in our formalism as in the usual BNF formalism, and will not be further explained. We denote non-terminal symbols in lower case in the grammar (e.g., `expr`).

The distinction between terminal symbols and markers is non-standard. Markers have no semantic value other than their presence; an example is `"abstract"`. Thus, the semantic value of a marker is a boolean value; it is either there, or it is not (note that this is different from a symbol such as the semi-colon, which has *no* semantic value whatsoever, and thus does not need to be included in an abstract syntax tree). Conversely, the semantic value of a *terminal symbol* is an arbitrary value; an example is `CLASS_NAME` (the structure of a terminal symbol may be defined by a regular expression; this is irrelevant as far as the abstract grammar is concerned). We denote markers in quotes (`"abstract"`), and terminal symbols in capitals (`CLASS_NAME`).

Each non-terminal symbol `aa` will have a single production in the grammar. Instances of `aa` in the AST will be represented by a class called `Aa`. The attributes of `Aa` will depend on the production for `aa` (see below).

A terminal symbol `xx` will be represented by a class `XX`. Every token class gets an attribute called `value`. The type of this attribute depends on the token; for most tokens it is `String*` (this is the default); however, if the grammar explicitely specifies a type for the value (in angular brackets, for example `REAL<double>`), this overrides the default. If the type of the `value` attribute it set to be empty, the token class does not get a value.

Finally, a marker will not be represented by a specialised class. Instead, a marker `"foo"` may *only* appear as an optional symbol in a production rule (`a ::= ... "foo"? ...`), and will appear as a boolean attribute `is_foo` in the class representing `aa` (`Aa`).

There are only two types of rules in the grammar. The first is the simplest, and list a number of alternatives for a non-terminal symbol `aa`:

```
aa ::= b | c | ... | z
```

Here, each of b, c, ..., z must be a single non-terminal symbol. This rule results in a (usually) empty `class Aa {}`, which acts as a superclass for the classes for b, c, ..., z. This reflects the semantics of the rule (a b *is* an a); if there are multiple rules `aa ::= c|...`, `b ::= c|...`, class C will inherit from both Aa and B. This type of rule is exemplified by the production for Statement in the grammar. There is one additional requirement for disjunction rules, which will be explained in the section on context resolution, below.

The second type is the most common:

```
aa ::= b c ... z
```

In this rule, each of the b, c, ..., z is an arbitrary symbol (non-terminal, terminal or marker), which may be optional (b?) or repeated (b* or b+). This type of rule must not include any disjunctions (b|c), and only single symbols can be repeated (no grouping). If a symbol b can be repeated, it will be represented by a specialised list class B_list (which inherits from the STL list class) in the tree. In addition, the symbols may be labeled (label:symbol). This does not add to the grammar structure, but explains the purpose of the symbol in the rule, and will be used for the name of the attribute of the corresponding class. The default name for each class attribute depends on the corresponding type: an attribute of type Variable_name (corresponding to a non-terminal Variable_name) will be called variable_name. The default name for an attribute of type Foo_list will be *foos*. However, as mentioned above, this can be overridden by specifying a label.

As an example, consider the rule for variable in the grammar.

```
Expr ::= ... | Variable | ... ;
Variable ::= Target? Variable_name array_indices:Expr?* ;
```

A Variable is an Expr, so that Variable is represented by the class shown below.

```
class Variable : virtual public Expr
{
public:
   Target* target;
   Variable_name* variable_name;
   Expr_list* array_indices;
}
```

A final note on combining * and ?. The construct (a*)? denotes an optional list of as. Thus, it will be represented by an A_list. If a list is specified, but empty, the list will simply contain no elements. If the list is not specified at all, the list will be NULL. This is used, for example, to distinguish between methods that contain no statements and abstract methods. Similarly, (a?)* is a (non-optional) list of optional as. Thus, this is a list, but elements of the list may be NULL. This is used for example to denote empty array indices (a[]) in the rule for Variable.

Context Resolution

We also derive the tree visitor API and tree transformation API from the grammar. The tree visitor API is very simple to derive; see the *Overview of the AST classes and transformation API* for an explanation.

The tree transformation API however is slightly more difficult to derive. The problem is to decide the signatures for the transform methods, or in other words, what can transform into what? For example, in the **phc** grammar for PHP, the transform for an if-statement should be allowed return a list of statements of any kind (because it is safe to replace an if-statement by a list of statements). Similarly, a binary operator should be allowed return any other expression (but not a list of them). For reasons that will become clear very soon, we call the process of deciding these signatures "context resolution".

Contexts

A context is essentially a use of a symbol somewhere in a (concrete) rule in the grammar. There are four possibilities. Consider:

```
concrete1 ::= ...
concrete2 ::= ...
concrete3 ::= ...
concrete4 ::= ...
concrete5 ::= ...
concrete6 ::= ...
abstract1 ::= concrete3 | concrete4
abstract2 ::= concrete5 | concrete6

some_concrete_rule ::= concrete1 concrete2* abstract1 abstract2*
```

then, based on the rule for some_concrete_rule, concrete1 occurs in the context (concrete1,concrete1,Single) - i.e., as a single instance of itself, concrete2 occurs in the context (concrete2,concrete2,List), i.e. as a list of instances of itself. The use of the abstract1 class leads to a number of contexts:

```
(abstract1,abstract1,Single)
(concrete3,abstract1,Single)
(concrete4,abstract1,Single)
```

And finally, the use of abstract2* yields to the contexts

```
(abstract2,abstract2,List)
(concrete5,abstract2,List)
(concrete6,abstract2,List)
```

These contexts essentially mean that an instance of concrete5 can be replaced by any number of any (concrete) instance of "abstract2".

Reducing Contexts

If there are two or more conflicting contexts for a single symbol, we must resolve the contexts to their most specific (restrictive) form. For instance, for the **phc** grammar, this yields

```
(if,statement,List)
(CLASS_NAME,CLASS_NAME,Single)
(INTERFACE_NAME,INTERFACE_NAME,Single)
```

So, a context is a triplet (symbol,symbol,multiplicity), where the symbols are terminal or non-terminal symbols, and the multiplicity is either Single, Optional, List, OptionalList or ListOptional (list of optionals). When reducing two contexts (`a`,`b`,`c`) (`a'`,`b'`,`c'`), we take the meet of `b` and `b'` (that is, the most general common subclass of `b` and `b'`, where more general means higher up in the inheritance hierarchy), and opt for the most restrictive Multiplicity (Single over Optional, Single over List, etc.). The general idea is that we want the most permissive context for a non-terminal that is still safe: if it is safe to replace an `a` by a list of `b`s *everywhere* in a tree, the context we want for `a` is (`a`, `b`, list).

To see the reason for taking the meet, consider this fragment of the **phc** grammar:

```
Expr ::= ... | BOOL
Cast ::= CAST Expr
Method_invocation ::= Target ...
Target ::= Expr | CLASS_NAME
```

The use of "expr" in the rule for cast leads to the context (BOOL,expr,Single) The use of "target" in the rule for method_invocation leads to the context (BOOL,target,Single). By taking the meet of "expr" and "target", this gives the context (BOOL,expr,Single). This means that it is always safe to replace a boolean by any other expression (but it is not always safe to replace a boolean by any other *target*).

In the case of CLASS_NAME, we have the contexts

```
(CLASS_NAME,class_name,Single)
(CLASS_NAME,target,Single)
```

The meet of class_name and target does not exist; hence this gives the context

```
(CLASS_NAME,CLASS_NAME,Single)
```

That is, the only safe transformation for CLASS_NAME is from CLASS_NAME to CLASS_NAME.

To be precise about the “most specific” multiplicity, here is a Haskell definition that returns the meet of two multiplicities:

```
meet_mult :: Multiplicity -> Multiplicity -> Multiplicity
meet_mult a b | a == b = a
meet_mult Single _ = Single
meet_mult List Optional = Single
meet_mult List OptList = List
meet_mult List ListOpt = List
meet_mult Optional OptList = Single
meet_mult Optional ListOpt = Optional
meet_mult OptList ListOpt = List
meet_mult a b = meet_mult b a  -- meet is commutative
```

Resolution for Disjunctions

We cannot deal with this situation:

```
s ::= a
a ::= b | c
d ::= b
```

```
e ::= c*
```

This grammar leads to the following contexts:

```
(a,a,Single)
(b,a,Single)
(b,b,Single)
(c,a,Single)
(c,c,List)
```

Resolving these contexts lead to

```
(a,a,Single)
(b,b,Single)
(c,c,List)
```

However, this is incorrect, because this indicates that an `a` will only be replaced by another, single, `a`; but a `c` (which is an `a`) will in fact return a list of `c`s. The problem is that the non-terminals in the rule for `a` have a different multiplicity in their contexts (single for `b`, list for `c`). `maketea` disallows this; if this happens in a grammar, `maketea` will exit with a "cannot deal with mixed multiplicity in disjunction" error.

Otherwise, for a rule `a ::= b1 | b2 | ...`, if the multiplicity of `a` is list, and the multiplicities of all the `b`s are lists, the multiplicity for `a` will be list; if the multiplicity of all the `b`s is single, the multiplicity for `a` will be set to single (independent of the original multiplicity for `a`).

III. Development guide

Chapter 14. Porting and Packaging

We need porters, packagers and maintainers

Now that **phc** has a plugin architecture, it is no longer necessary for users to integrate their source with it. As a result, it is much more useful to package **phc** and integrate it within various distributions' package management systems. If you are interested in packaging **phc** for your favourite OS, please contact us (http://www.phpcompiler.org/mailinglist.html).

Currently, **phc** runs on x86 Linux, and is mostly tested using Ubuntu. If you have access to other machines, architectures or operating systems, and would be willing to test **phc** on it, please contact us (http://www.phpcompiler.org/mailinglist.html).

Packaging hints

Do not strip the binaries. Since the plugins use `dlopen()`, and link dynamically against the **phc** binary, the plugins will not work unless the symbol information is available.

Test suite

phc is shipped with its tests, which can be run after compilation:

```
make test
```

You can also add your own code to `test/subjects/3rdparty`, and test it by running:

```
make long-test
```

to run the entire suite. Note that many tests still fail. Please submit the results to us on the mailing list (http://phpcompiler.org/mailinglist.html).

phc packages

See the downloads page (http://www.phpcompiler.org/src) for existing packages. In addition, we're looking for people to create and/or maintain packages for more systems, including Debian/Ubuntu (especially Debian/Ubuntu), Gentoo, Slackware, Darwin and Solaris.